

Approaches and Tools for Understanding and Improving the Performance of Scientific Applications

Boyana Norris

Computer Scientist

Argonne National Laboratory

<http://www.mcs.anl.gov/~norris>

Acknowledgments

- Funding:
 - DOE Scientific Discovery through Advanced Computing (SciDAC) program
 - National Science Foundation
 - Tech-X Corporation

- Collaborators:
 - Dr. S. H. K. Narayanan and P. Hovland, Argonne National Laboratory
 - Dr. Albert Hartono, Reservoir Labs
 - Prof. P. (Saday) Sadayappan, Ohio State University
 - Prof. William Gropp, University of Illinois at Urbana-Champaign
 - Prof. Elizabeth Jessup and Prof. Jeremy Siek, University of Colorado at Boulder

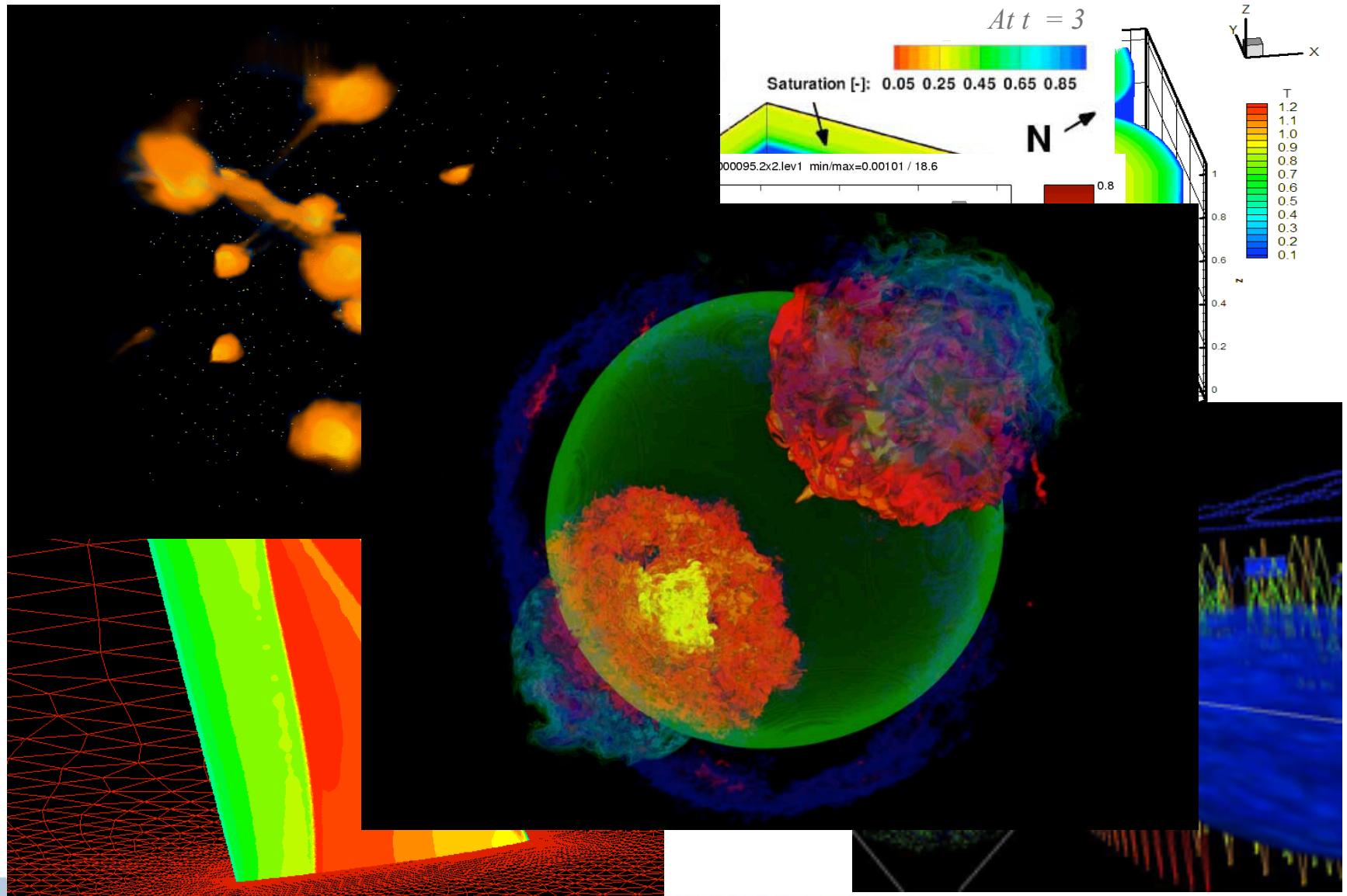


Outline

- Motivation
- Part I: Understanding performance
 - Generating performance models from source code
- Part II: Automatically optimizing performance
 - Simple Example
 - Approach: Annotation-driven empirical tuning
 - Performance Experiments
- Summary



Application examples



Part I: Understanding performance

- Three main approaches to analyzing performance:
 - Empirical
 - Static (source or object code-based)
 - Analytical
- In this talk:
 - Focus on a hybrid static/empirical approach for generating performance models of computational kernels
 - Estimate *upper performance bounds*



What are performance bounds?

- Performance bounds give the upper limit in performance that can be expected for a given application on a given system
- Current approaches:
 - Fully algorithmic
 - Ignores machine information
 - Theoretical peak of the machine
 - Based only on available floating point units
 - Fully dynamic with profiling
 - Profiling overheads, time consuming
- Our approach
 - Application signatures + Architectural Information => Bounds



How does this help us?

- Determines the efficiency of the application
 - Efficiency = Observed Performance / Performance Bound
- Provides insight on how to improve performance
 - Low efficiency:
 - Re-engineer/tune the implementation
 - High efficiency but low performance:
 - Change algorithm or architecture



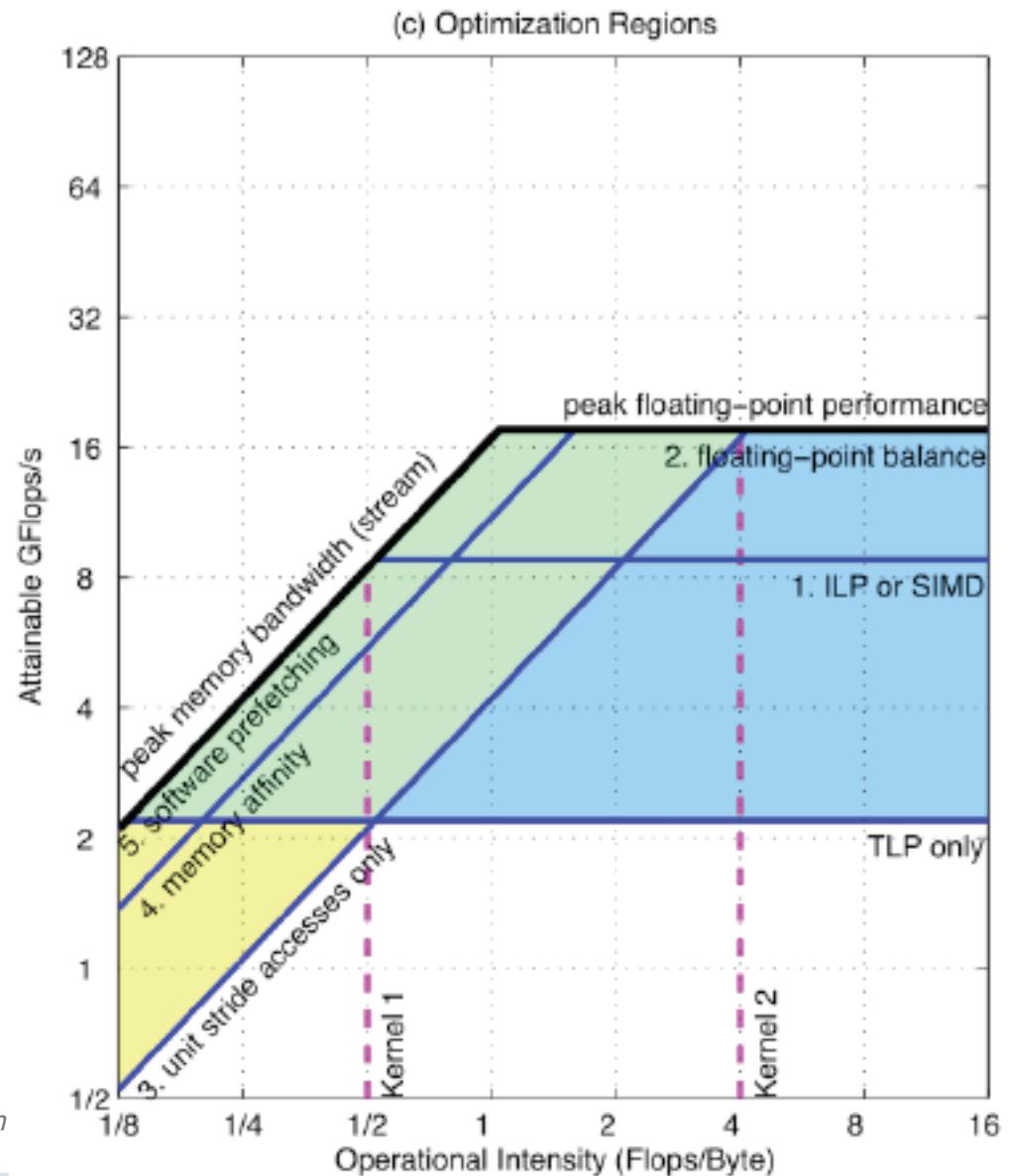
Related work

- Gropp, Kaushik et al.
 - Provided the motivation for this work
 - Determined the performance bound by counting the number of arithmetic operations and memory accesses per statement.
 - Performed the analysis on a representation of the source code
- Vuduc et al.
 - Manual bounds estimation for sparse $A^T A x$ computation kernel



Related work : Roofline model

- Shows the ‘region’ of possible application performance
- Determines how optimizations affect application performance
- The performance space is determined through runtime means.



Source : S. Williams, A. Waterman and D. Patterson
CACM 2009



So what is PBound?

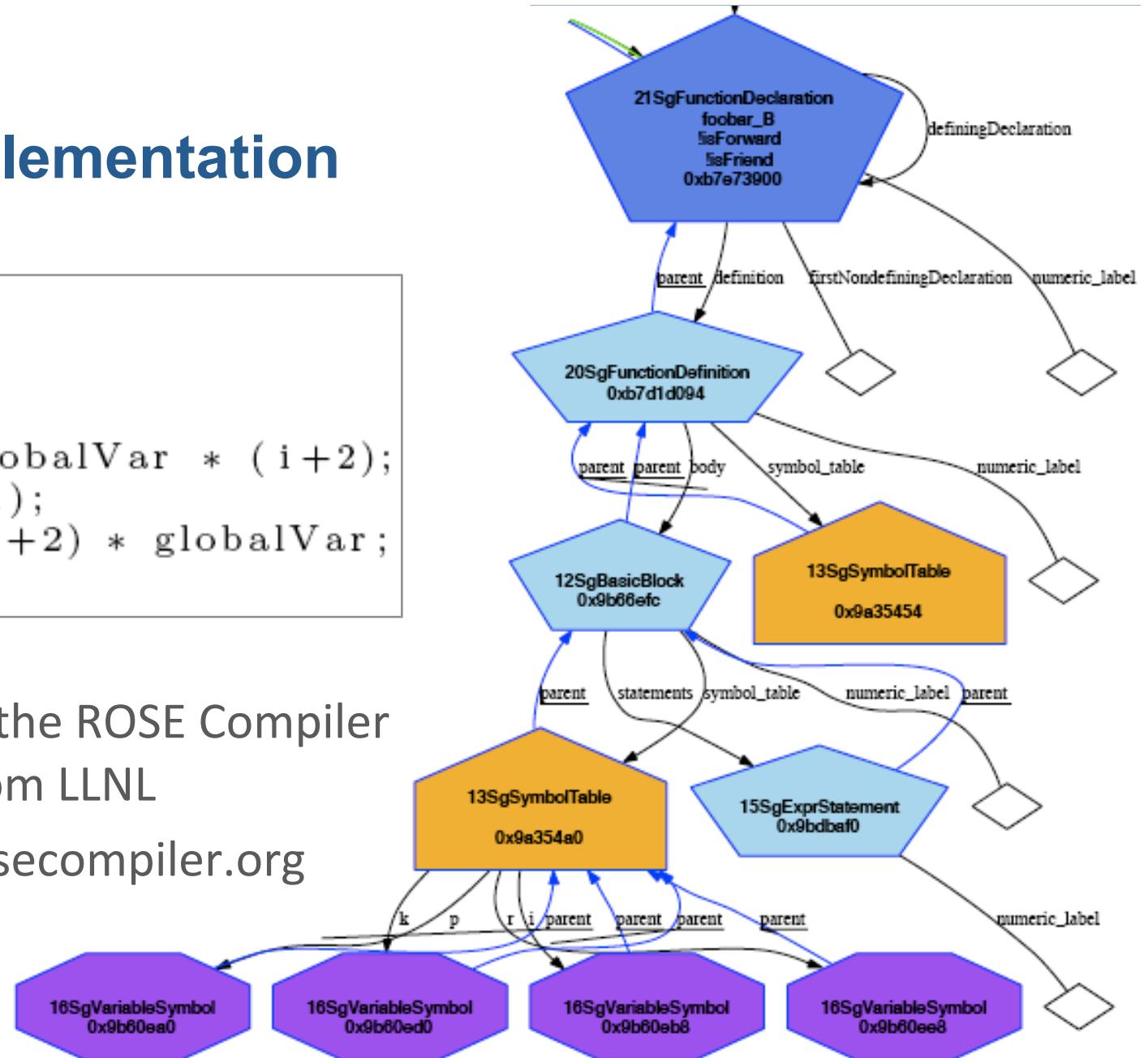
- ❑ Our tool to generate performance bounds from source code
- ❑ Uses static (source code) analysis
 - Produces parameterized closed-form expressions expressing the computational and data load/store requirements of application kernels.
- ❑ Coupled with architectural information
 - Produces upper bounds on the performance of the application



Pbound implementation

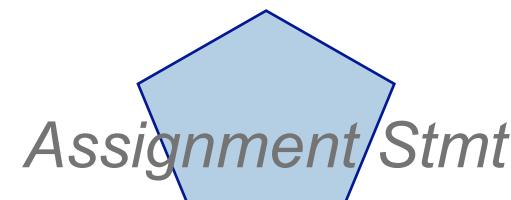
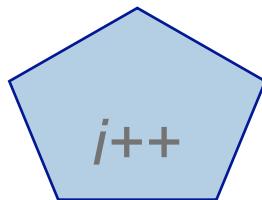
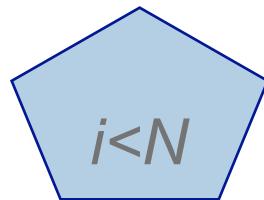
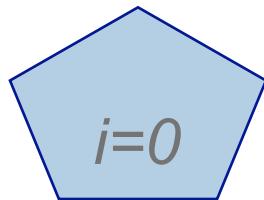
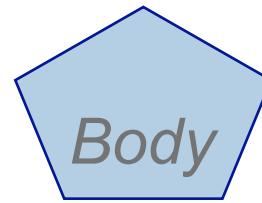
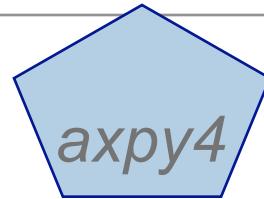
```
void foobar_B()
{
    int p;
    int i = 4;
    int k = globalVar * ( i + 2 );
    p = foo( k );
    int r = ( p + 2 ) * globalVar;
}
```

- Built on top of the ROSE Compiler Framework from LLNL
- <http://www.rosecompiler.org>



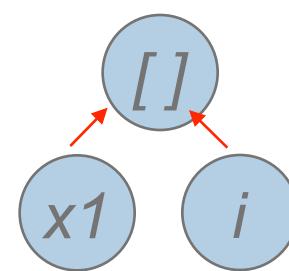
AST Representation

```
void axpy4(int n, double *y, double a1, double *x1, double a2,  
          double *x2, double a3, double *x3, double a4, double *x4) {  
    register int i;  
    for (i=0; i<=n-1; i++)  
        y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i];  
}
```



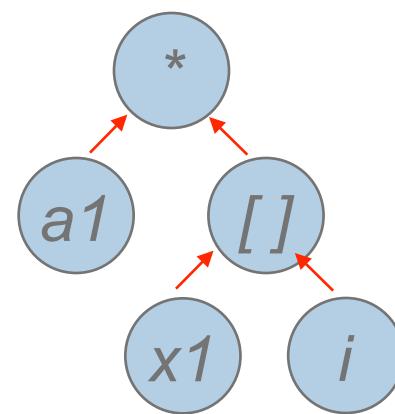
A Closer Look at the AST

Assignment Stmt



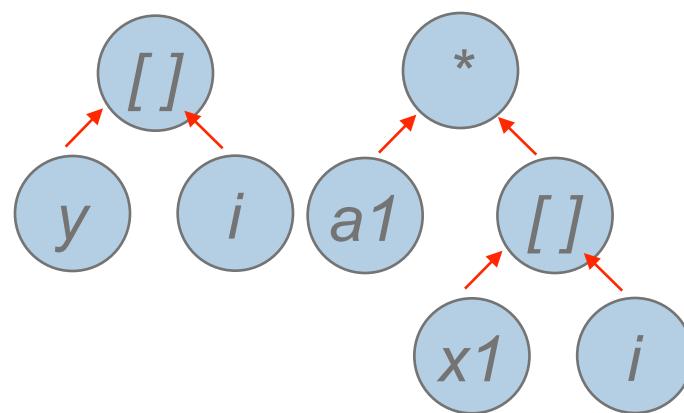
A Closer Look at the AST

Assignment Stmt



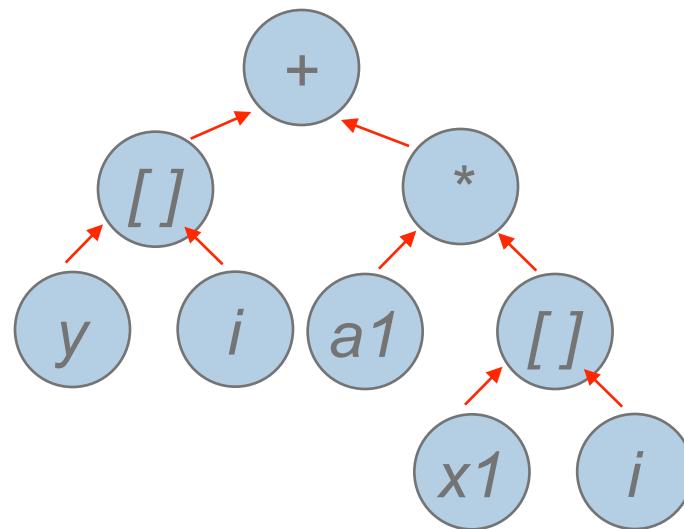
A Closer Look at the AST

Assignment Stmt



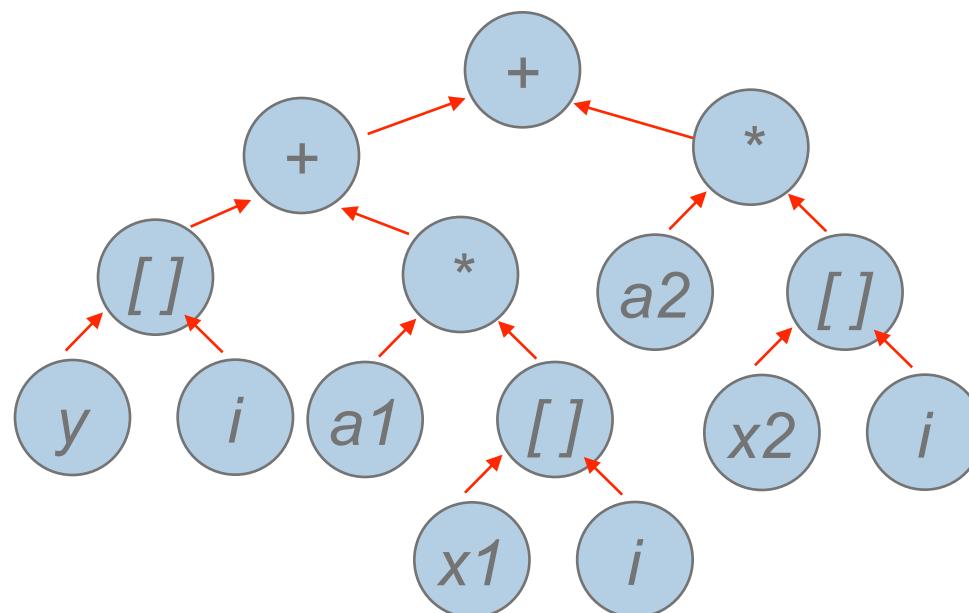
A Closer Look at the AST

Assignment Stmt



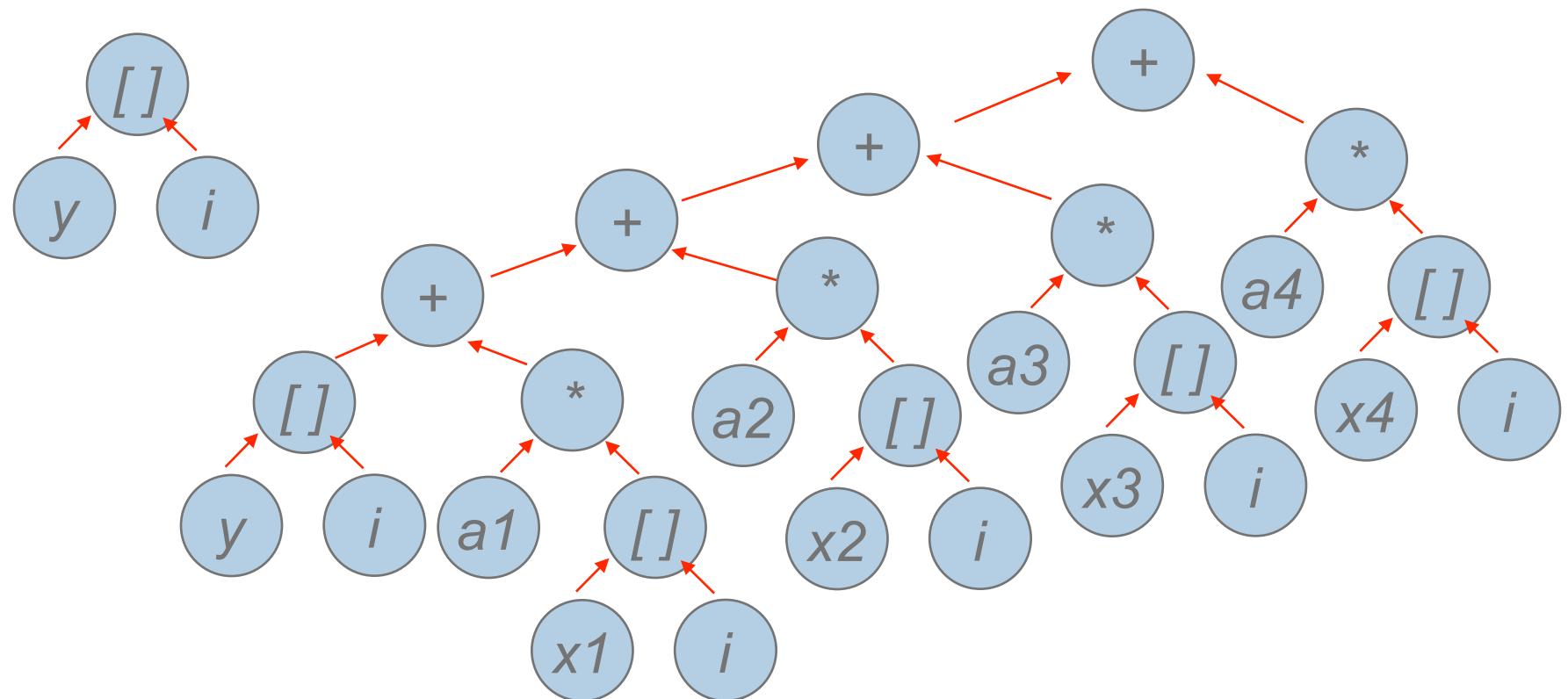
A Closer Look at the AST

Assignment Stmt

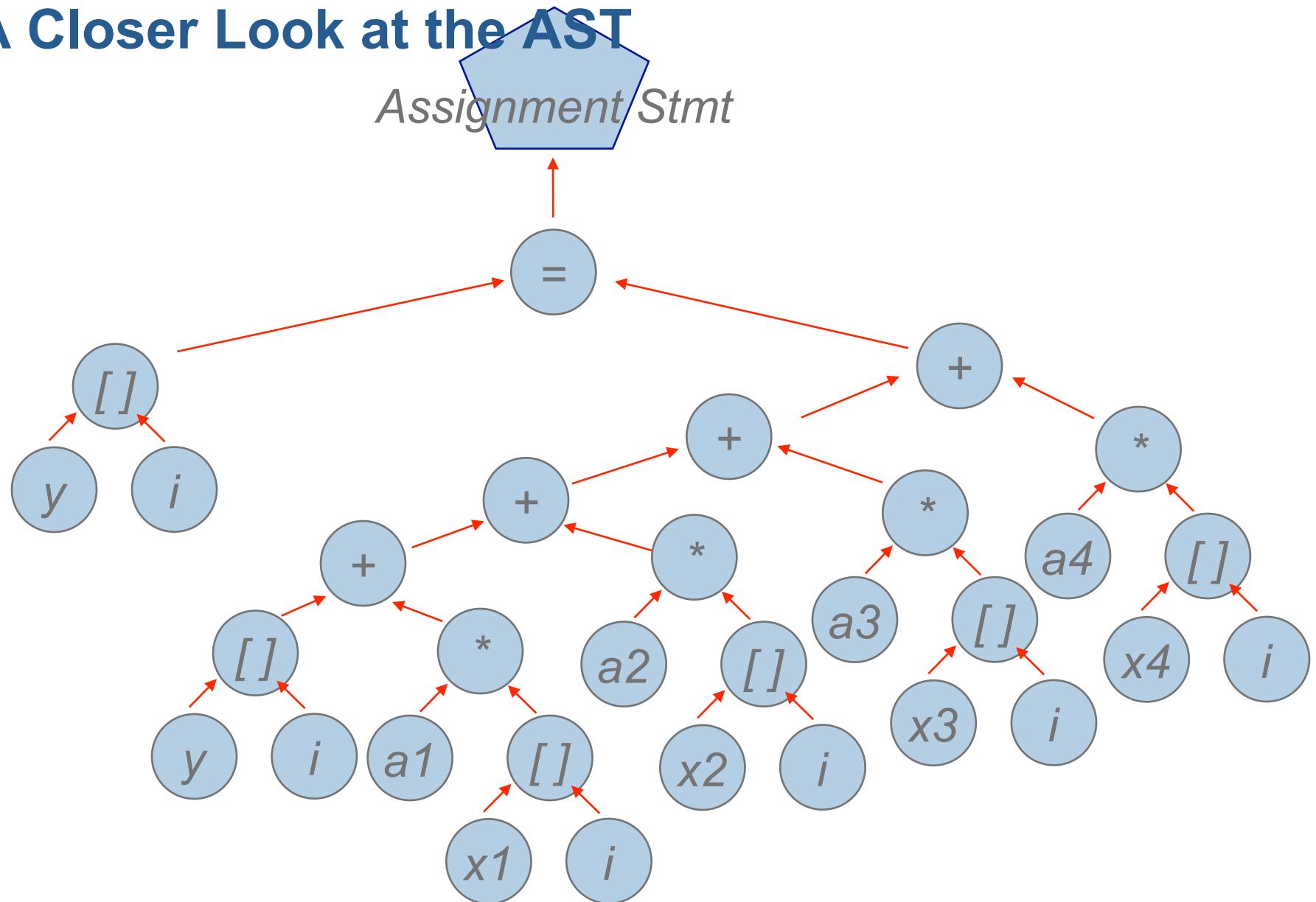


A Closer Look at the AST

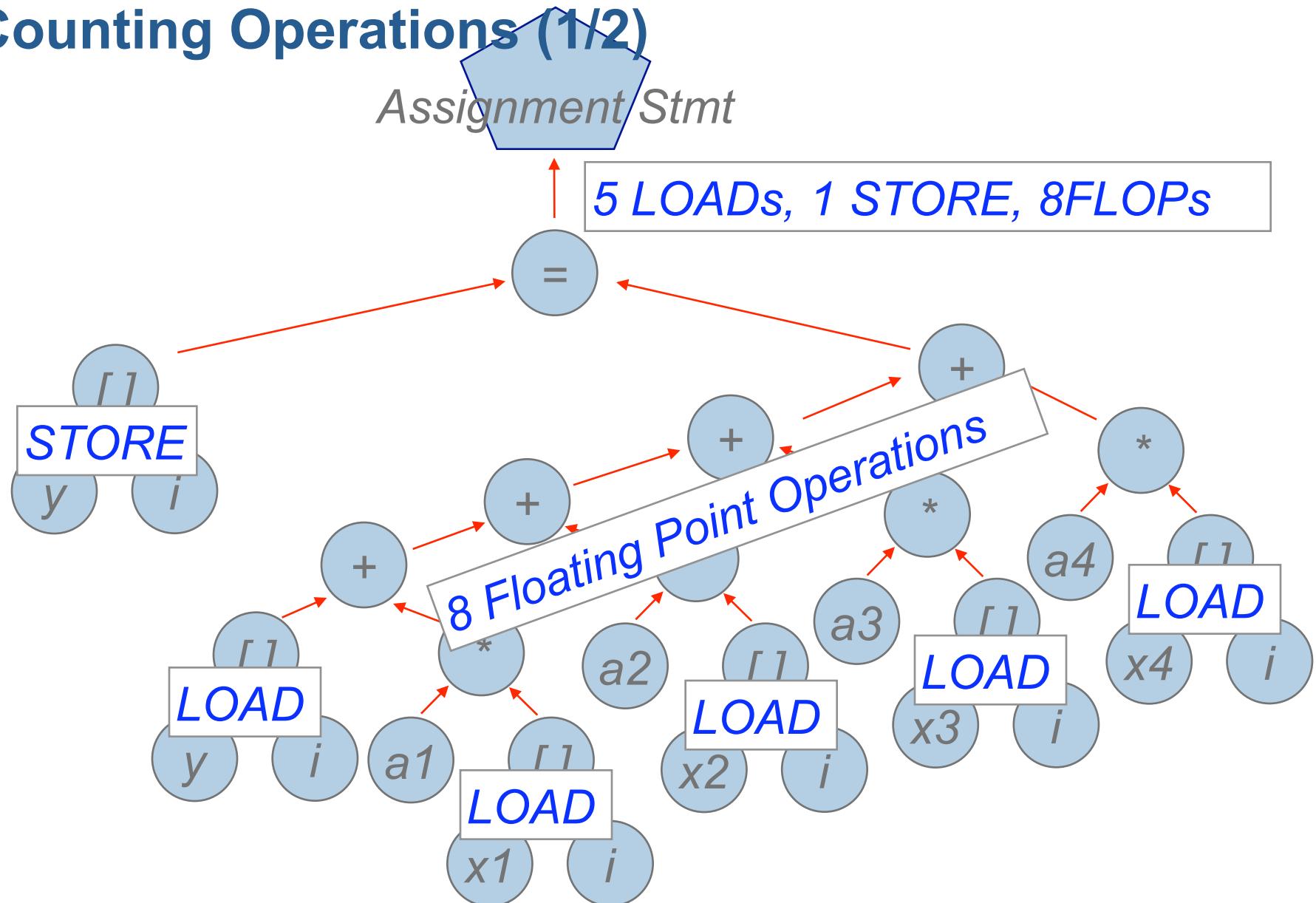
Assignment Stmt



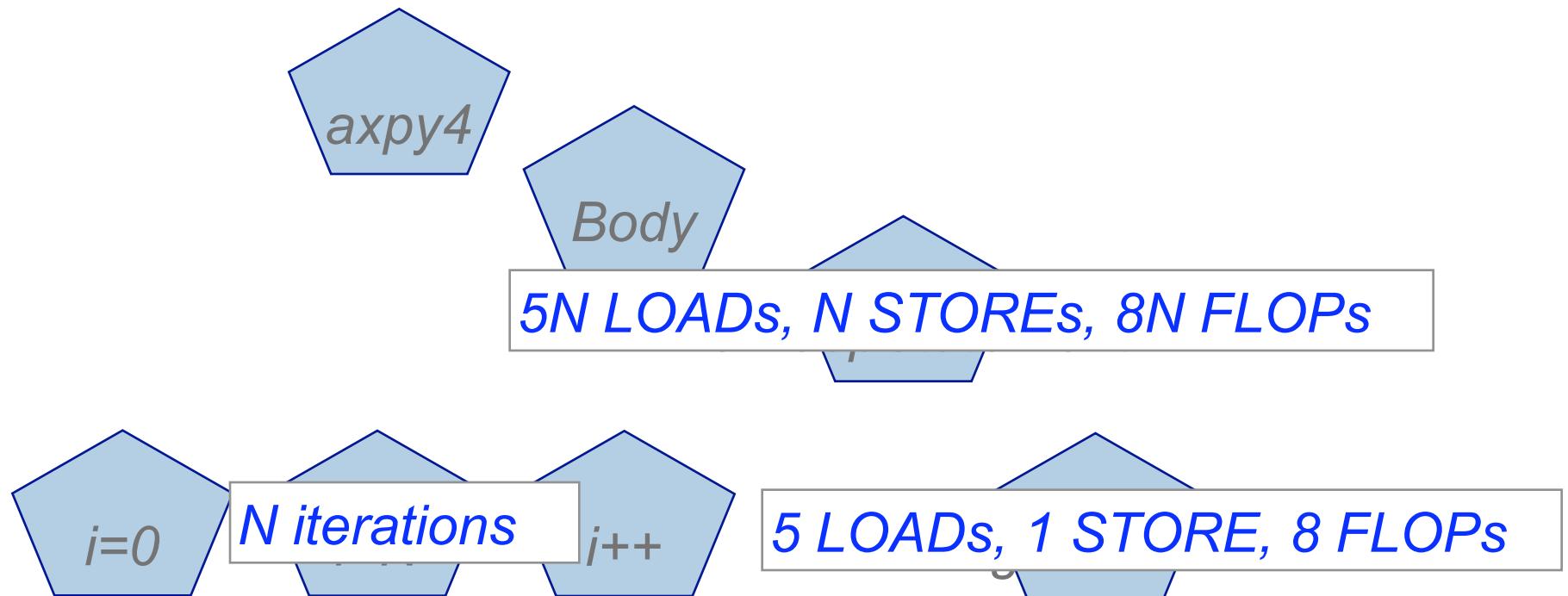
A Closer Look at the AST



Counting Operations (1/2)



Counting Operations (2/2)



Generated output

```
void axpy4(int n, double *y, double a1, double *x1,
double a2, double *x2, double a3, double *x3, double
a4, double *x4) {
    register int i;
    for (i=0; i<=n-1; i++)
        y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i];
}
```



```
#include "pbounds_list.h"
void axpy4(int n, double *y, double a1, double *x1,
double a2, double *x2, double a3, double *x3, double
a4, double *x4){
#endif pbounds_log
pboundsLogInsert("axpy.c@6@5",8,0,40 * ((n - 1) + 1) +
32,     8 * ((n - 1) + 1),3 * ((n - 1) + 1) + 1,4 *
((n - 1) + 1));
#endif
}
```



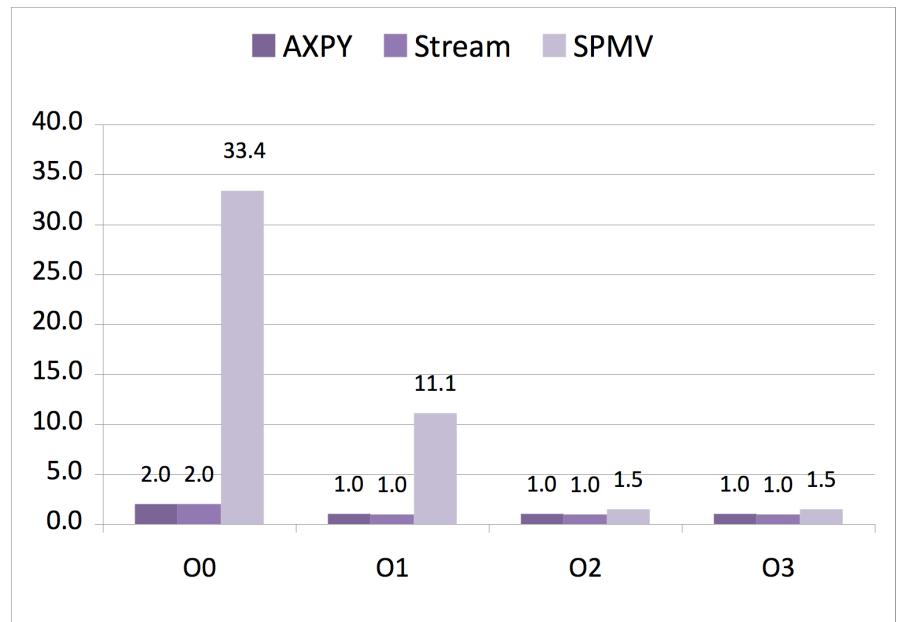
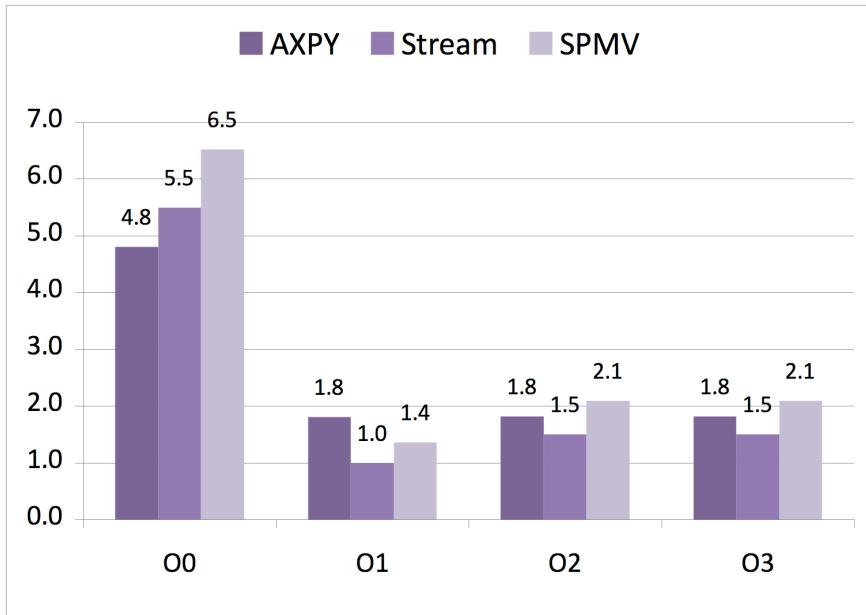
Architecture details

- SIMD
 - Specify vector length
 - Requirements
 - Types of all operations involved must be the same
 - Alignment of all values must be the same
 - Loop increment/decrement must be 1
- Fused Operations
 - fp_mul_add
 - fp_mul_sub
 - fp_div_add
 - fp_div_sub

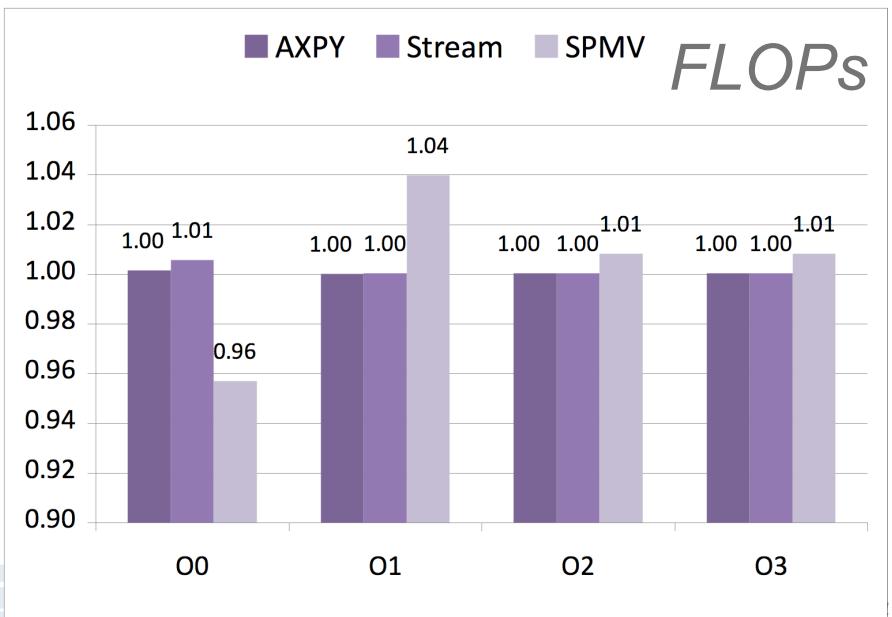


Example: Xeon

Loads

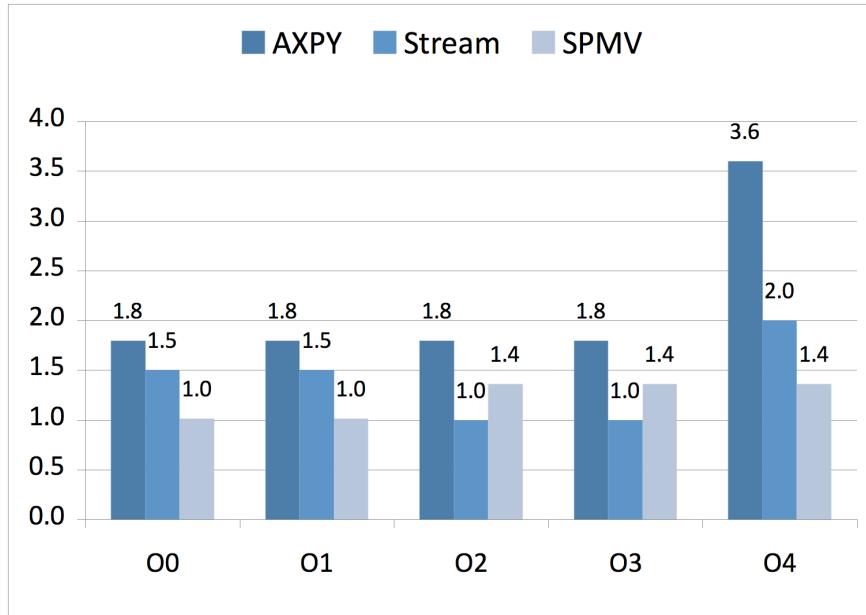


- Tuning and Analysis Utilities from UOregon used for profiling
- Bars show ratio of observed value to predicted value

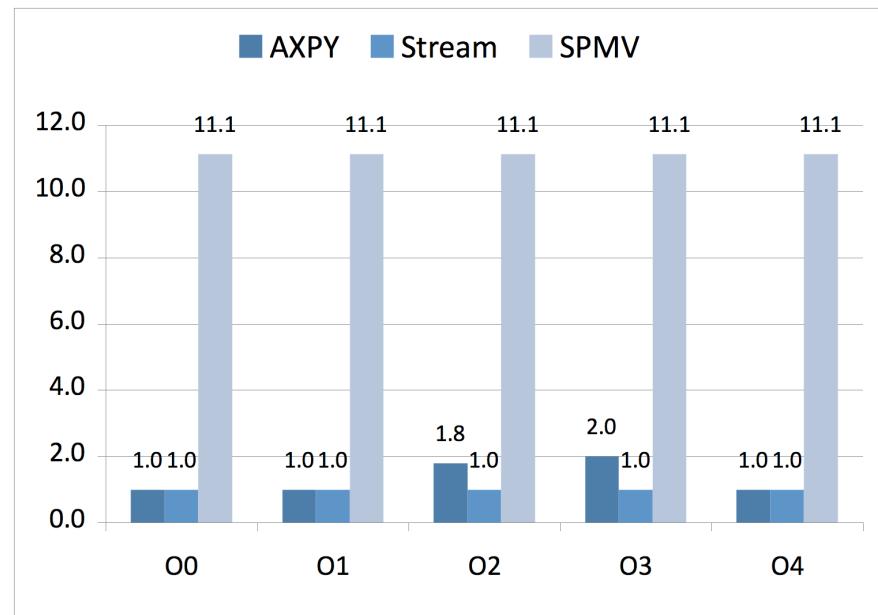


Example: BG/P

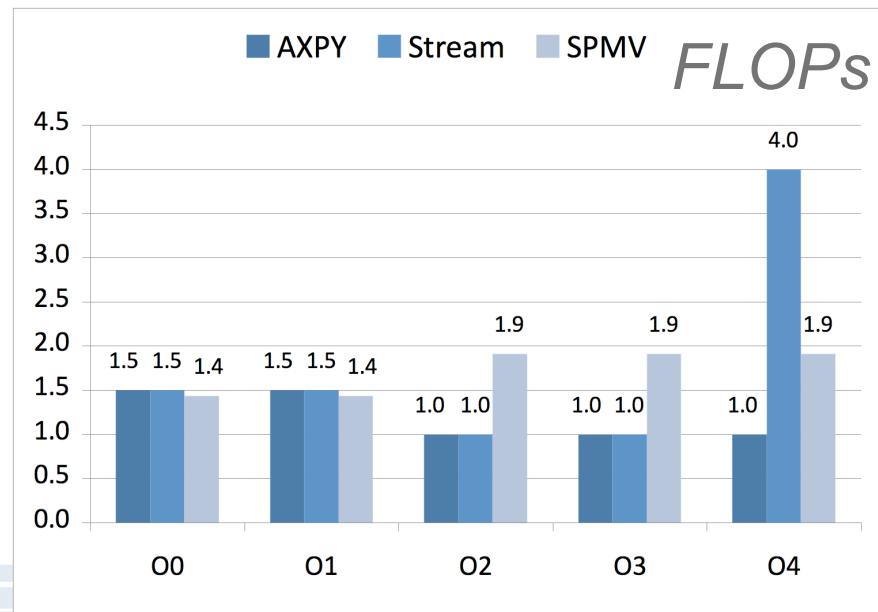
Loads



Stores



- Used the Hardware Performance Monitor Toolkit



Using the data - Generating Performance Bound

$$UpperBound = N * P_m$$

- ❑ N - Ratio of FLOPs to Bytes as determined by PBound
- ❑ P_m - Peak Reads Bandwidth determined by LMBench

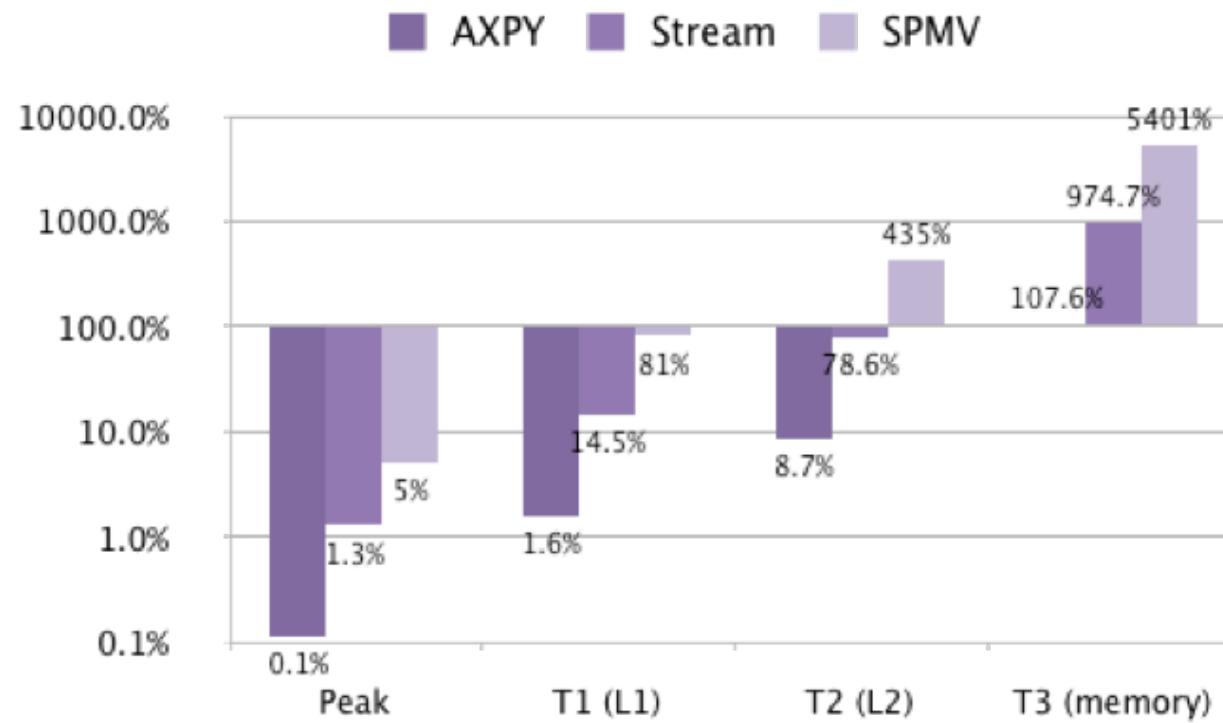
	N (FLOPs/Byte)	UB GFLOP/s	UB/Pc
Axpy	1.60	6.01	53.70%
STREAM	1.00	3.76	33.56%
SPMV	0.67	2.51	22.38%



Using the data: Predicting wallclock time

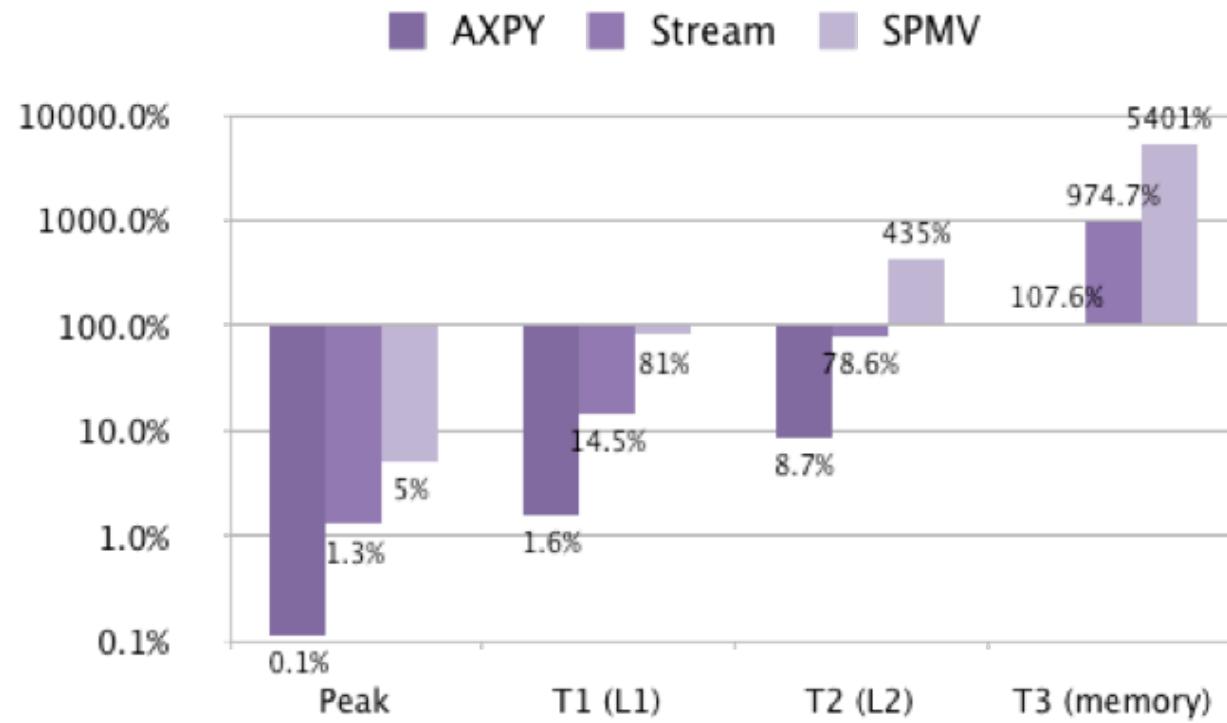
- Expected time based on peak FLOP counts
 $= 4\text{FLOPs/cycle} * 2.8 * 1\text{e}9 \text{ cycles/second}$

Ratio of Predicted time vs. Measured Time



Using the data – Predicting wallclock time

- ❑ Models assume that all the data is present in L1 cache, L2 cache, L3 cache, main memory
- ❑ Applications were determined to be memory bound by Pbound and LMBench
 - The degree of affects the result



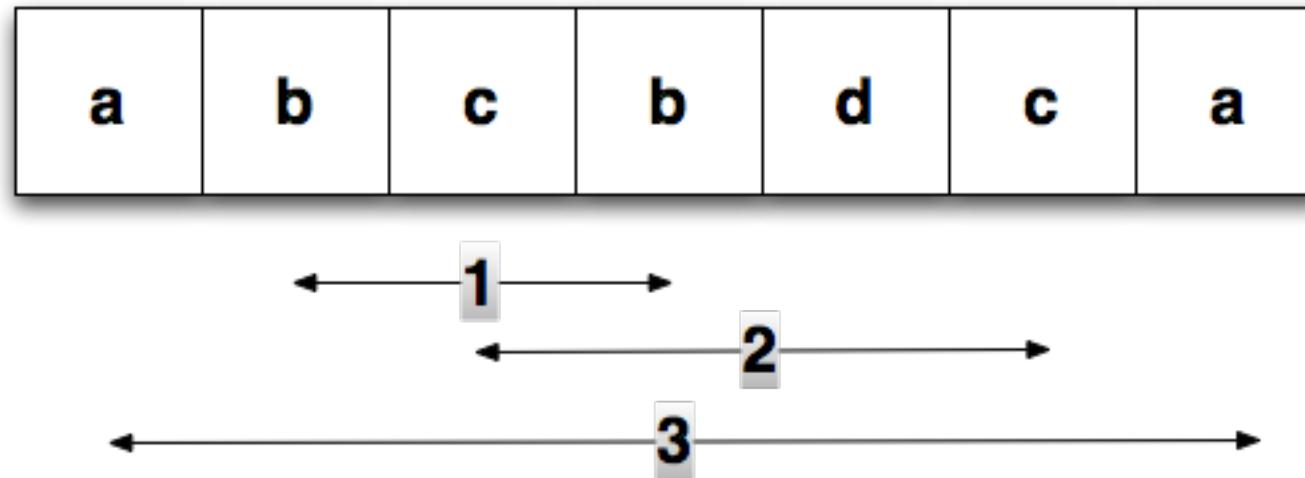
Ongoing Work - Aggressive Slicing

- ❑ Use Alias Analysis to determine unique/non-unique memory accesses.
- ❑ Use dataflow analyses to determine statements needed for particular target statements.
- ❑ Remove all other statements.



Ongoing work: Reuse distance analysis

- ❑ Based on estimating cache reuse distance



- ❑ There are two approaches currently
 - Trace based
 - Static analysis based
- ❑ Our Approach
 - Static Analysis based approach but takes advantage of one sliced execution.



Ongoing work: Analyzing parallel (MPI) codes

- ❑ Goal : Communication matrix to contain parametrized expressions of bytes of data sent from one process to another.
- ❑ Initially want to support point-to-point communication
- ❑ Approach
 - Calculate counts of different control flow paths.
 - Generate parametrized expressions.
 - Slice away statements not affecting expressions.
 - Run sliced code - does not actually perform any communication.



Part II: Optimizing performance

Challenges:

- Abstractions provided by HPC programming languages or environments are not close enough to the scientist
 - Desire for high levels of abstraction *and* high performance
 - Success of libraries that offer domain-specific abstractions
 - Data structures
- Lack of performance portability
 - Tuning for a particular architecture potentially hinders performance on different architectures
- Compiler optimizations insufficient
 - Compiler lag for new architectures
 - Loss of information when using general-purpose languages
- High performance often means decreased developer productivity



- How do we achieve good performance on different architectures without sacrificing portability, readability, and maintainability of software?

One answer: Use a simpler language (restricted or domain-specific), e.g., simplified C or MATLAB, and figure out how to optimize its performance.



Numerical Kernel Examples

- For example, dense matrix-matrix product (DGEMM)
 - Example code from “Superscalar GEMM-based Level 3 BLAS”, Gustavson et al. (next slide)
- BLAS-like operations for which there are no vendor-optimized libraries
 - Jacobian and Hessian accumulation code in AD-generated derivative computations
 - Tensor products
 - Composed linear algebra operations, e.g., $y = a(Ax) + b(Bx)$ where a and b are scalar variables, x and y are column vectors, and A and B are matrices
- PETSc code for sparse matrix operations
 - Manual optimizations include unrolling and use of registers
 - Code for diagonal matrix format is fast on cache-based systems but slow on vector systems
 - Too much code to rewrite by hand for new architectures



A Fast DGEMM (Sample)

```

SUBROUTINE DGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
                   BETA, C, LDC )
...
UISEC = ISEC-MOD( ISEC, 4 )
DO 390 J = JJ, JJ+UISEC-1, 4
DO 360 I = II, II+UISEC-1, 4
  F11 = DELTA*C( I,J )
  F21 = DELTA*C( I+1,J )
  F12 = DELTA*C( I,J+1 )
  F22 = DELTA*C( I+1,J+1 )
  F13 = DELTA*C( I,J+2 )
  F23 = DELTA*C( I+1,J+2 )
  F14 = DELTA*C( I,J+3 )
  F24 = DELTA*C( I+1,J+3 )
  F31 = DELTA*C( I+2,J )
  F41 = DELTA*C( I+3,J )
  F32 = DELTA*C( I+2,J+1 )
  F42 = DELTA*C( I+3,J+1 )
  F33 = DELTA*C( I+2,J+2 )
  F43 = DELTA*C( I+3,J+2 )
  F24 = DELTA*C( I+2,J+3 )
  F44 = DELTA*C( I+3,J+3 )
DO 350 L = LL, LL+LSEC-1
  F11 = F11 + T1( L-LL+1, I-II+1 )*
                 T2( L-LL+1, J-JJ+1 )
  F21 = F21 + T1( L-LL+1, I-II+2 )*
                 T2( L-LL+1, J-JJ+1 )
  F12 = F12 + T1( L-LL+1, I-II+1 )*
                 T2( L-LL+1, J-JJ+2 )
  F22 = F22 + T1( L-LL+1, I-II+2 )*
                 T2( L-LL+1, J-JJ+2 )
  F13 = F13 + T1( L-LL+1, I-II+1 )*
                 T2( L-LL+1, J-JJ+3 )
  F23 = F23 + T1( L-LL+1, I-II+2 )*
                 T2( L-LL+1, J-JJ+3 )
  F14 = F14 + T1( L-LL+1, I-II+1 )*
                 T2( L-LL+1, J-JJ+4 )
  F24 = F24 + T1( L-LL+1, I-II+2 )*
                 T2( L-LL+1, J-JJ+4 )
  F31 = F31 + T1( L-LL+1, I-II+3 )*
                 T2( L-LL+1, J-JJ+1 )
  F41 = F41 + T1( L-LL+1, I-II+4 )*
                 T2( L-LL+1, J-JJ+1 )
  F32 = F32 + T1( L-LL+1, I-II+3 )*
                 T2( L-LL+1, J-JJ+2 )
  F42 = F42 + T1( L-LL+1, I-II+4 )*
                 T2( L-LL+1, J-JJ+2 )
  F33 = F33 + T1( L-LL+1, I-II+3 )*
                 T2( L-LL+1, J-JJ+3 )
  F43 = F43 + T1( L-LL+1, I-II+4 )*
                 T2( L-LL+1, J-JJ+3 )
  F34 = F34 + T1( L-LL+1, I-II+3 )*
                 T2( L-LL+1, J-JJ+4 )
  F44 = F44 + T1( L-LL+1, I-II+4 )*
                 T2( L-LL+1, J-JJ+4 )
CONTINUE
...
*   End of DGEMM.
*
END

```

Why not just

```

do i=1,n
  do j=1,m
    c(i,j) = 0
    do k=1,p
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    enddo
  enddo
enddo
c = a * b

```

Or even

Our Approach: Annotation-Based Performance Tuning

- Meta-language for expressing
 - Semantic information
 - Performance tuning options
- Exploit (domain-specific) knowledge; alleviate costs associated with generality



Example

```
void axpy_4(int n, double *y, double a1, double *x1, double a2,
            double *x2, double a3, double *x3, double a4, double *x4)
{
    int i;

    for (i = 0; i < n; i++)
        y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
}
```



Example

```
void axpy_4(int n, double *y, double a1, double *x1, double a2,
            double *x2, double a3, double *x3, double a4, double *x4)
{
    int i;
    /*@ begin Align (x1[],x2[],x3[],x4[],y[]) @*/
    for (i = 0; i < n; i++)
        y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
    /*@ end @*/
}
```



Example

```
void axpy_4(int n, double *y, double a1, double *x1, double a2,
            double *x2, double a3, double *x3, double a4, double *x4)
{
    int i;
    /*@ begin Align (x1[],x2[],x3[],x4[],y[]) @*/
    /*@ begin Loop (
        transform Unroll(ufactor=UF, parallelize=PAR, simd=SIMD_TYPE)
        for (i=0; i <= N-1 ; i++)
            y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
    ) @*/
    for (i = 0; i < n; i++)
        y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
    /*@ end @*/
    /*@ end @*/
}
```



Example (cont.)

```
spec axpy4_tune_spec {
    def build {
        arg build_command =
            'mpixlc_r -O3 -qstrict -qhot -qsmp=omp:noauto';
        arg batch_command =
            'qsub -n 128 -t 20 --env "OMP_NUM_THREADS=4"';
        arg status_command = 'qstat';
        arg num_procs = 128;
    }
    def performance_counter {
        arg method = 'basic timer';
        arg repetitions = 10000;
    }
    def performance_params {
        param UF[] = range(1,33);
        param PAR[] = [True,False];
        param SIMD_TYPE[] = ['none','xlc'];
        constraint simd_unroll_factor = (SIMD_TYPE=='none'
            or UNROLL_FAC_IN%2==0);
    }
    def input_params {
        param N[] = [10,100,1000,10**4,10**5,
                    10**6,10**7];
    }
}
```

```
def input_vars {
    decl dynamic double y[N] = 0;
    decl dynamic double x1[N]
        = random;
    decl double a1 = random;
    decl double a2 = random;
    # ... omitted ...
}

def search {
    arg algorithm = 'Exhaustive';
    arg time_limit = 20;
}
```



Generated Code Fragment (full code > 100 lines)

```
void axpy_4(int n, double *y, double a1, double *x1, double a2,
           double *x2, double a3, double *x3, double a4, double *x4)
{
    int i;
    /*@ begin Align (x1[],x2[],x3[],x4[],y[]) @*/
    #pragma disjoint (*x1, *x2, *x3, *x4, *y)
    if (((((int)(x1)|(int)(x2)|(int)(x3)|(int)(x4)|(int)(y)) & 0xF) == 0) {
        __alignx(16,x1);
        __alignx(16,x2);
        __alignx(16,x3);
        __alignx(16,x4);
        __alignx(16,y);

        /*@ begin Loop (
            transform Unroll(ufactor=UF, parallelize=PAR)
            for (i=0; i<=N-1; i++)
                y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
        ) @*/
        #if ORIGLOOP
            for (i = 0; i < n; i++)
                y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
        #else
        {
            for ((i = 0); (i <= (n - 1) - 3); (i += 4)) {
                double _Complex _i_541, _i_542, _i_543, _i_544, _i_545, _i_555, _i_556,
                      _i_664, _i_665;
                _i_541 = __lfpd(&y[i]);
                _i_542 = __lfpd(&x4[i]);
                _i_543 = __fxcpmadd(_i_541, _i_542, a4);
                _i_544 = __lfpd(&x3[i]);
                _i_545 = __fxcpmadd(_i_543, _i_544, a3);
                _i_555 = __lfpd(&x2[i]);
                _i_556 = __fxcpmadd(_i_545, _i_555, a2);
                _i_664 = __lfpd(&x1[i]);
                _i_665 = __fxcpmadd(_i_556, _i_664, a1);
                __stfpd(&y[i], _i_665);
            }
            {
                double _Complex _i_541, _i_542, _i_543, _i_544, _i_545, _i_555, _i_556,
                      _i_664, _i_665;
                _i_541 = __lfpd(&y[(i + 2)]);
                _i_542 = __lfpd(&x4[(i + 2)]);
                _i_543 = __fxcpmadd(_i_541, _i_542, a4);
                _i_544 = __lfpd(&x3[(i + 2)]);
                _i_545 = __fxcpmadd(_i_543, _i_544, a3);
                _i_555 = __lfpd(&x2[(i + 2)]);
                _i_556 = __fxcpmadd(_i_545, _i_555, a2);
                _i_664 = __lfpd(&x1[(i + 2)]);
                _i_665 = __fxcpmadd(_i_556, _i_664, a1);
                __stfpd(&y[(i + 2)], _i_665);
            }
        }
    }
}
```



More Complex Annotation Example (Matrix-Matrix Product)

```
/*@ begin Loop(
    transform Composite(
        tile = [('i',T1_I,'ii'),('j',T1_J,'jj'),('k',T1_K,'kk'),
                ('ii','i'),T2_I,'iii'),((jj,'j'),T2_J,'jjj'),((kk,'k'),T2_K,'kkk')],
        permut = [PERMUTS],
        arrcopy = [(ACOPY_A,'A[i][k]',[(T1_I if T1_I>1 else T2_I),(T1_K if T1_K>1 else T2_K)],'_copy'),
                    (ACOPY_B,'B[k][j]',[(T1_K if T1_K>1 else T2_K),(T1_J if T1_J>1 else T2_J)],'_copy'),
                    (ACOPY_C,'C[i][j]',[(T1_I if T1_I>1 else T2_I),(T1_J if T1_J>1 else T2_J)],'_copy')],
        unrolljam = [('k',U_K),('j',U_J),('i',U_I)],
        scalarreplace = (SCREP, 'double', 'scv_'),
        vector = (VEC, ['ivdep','vector always']),
        openmp = (OMP, 'omp parallel for private(iii,jjj,kkk,ii,jj,kk,i,j,k,A_copy,B_copy,C_copy)')
    )
)

for(i=0; i<=M-1; i++)
    for(j=0; j<=N-1; j++)
        for(k=0; k<=K-1; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
) @*/
```



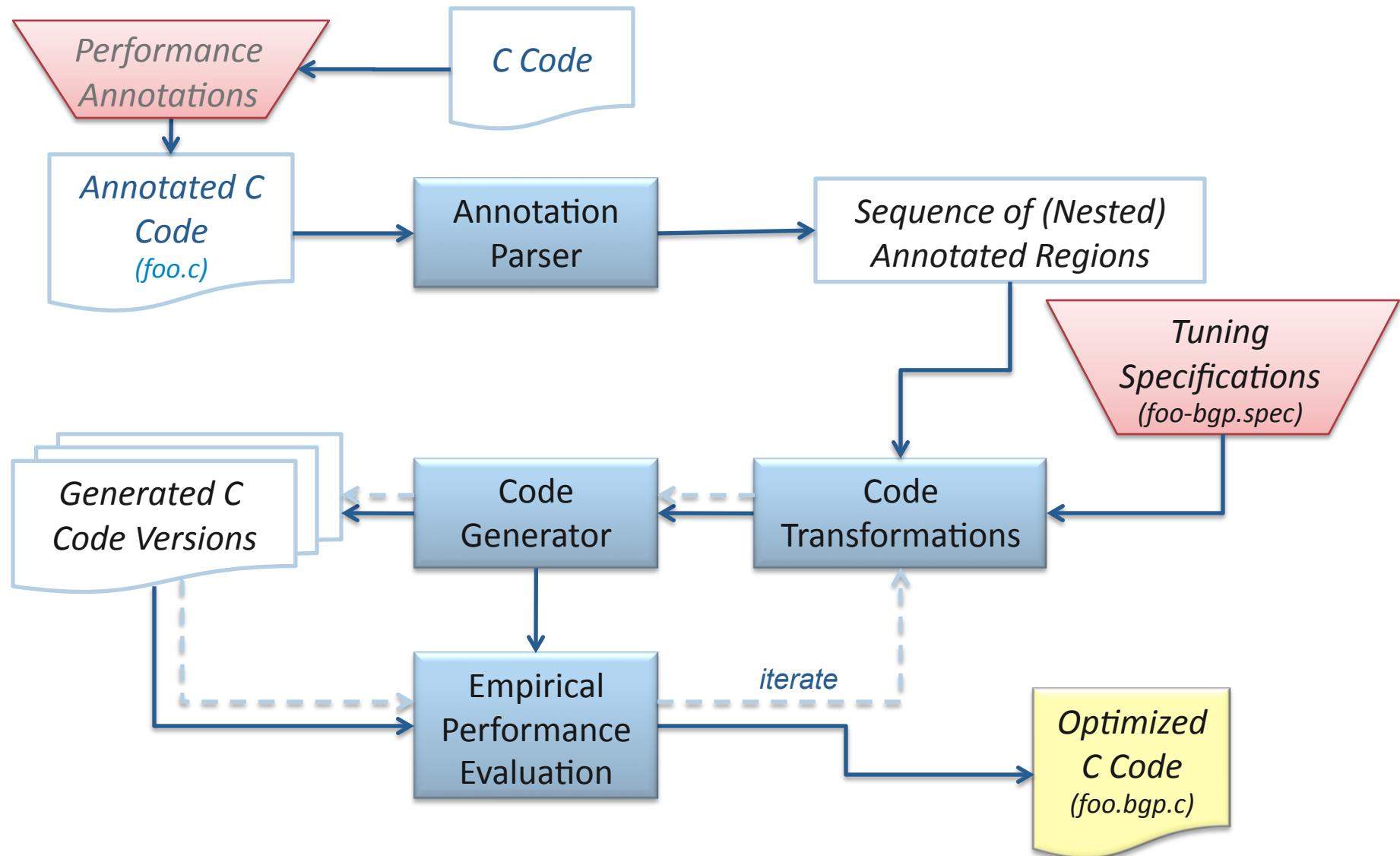
Approach (cont.)

- Design a flexible system to enable *non-experts* to define annotations that direct performance-tuning source transformations
- Goals
 1. Support source transformations between domain-specific or special-purpose languages and general-purpose languages (C/C++ and Fortran).
 2. Provide an effective approach for addressing performance issues by permitting (but not requiring) access by the application scientist to low-level details.
 3. Improve performance portability by abstracting platform-specific low-level optimization code.
 4. Preserve application investment in current languages.
- Software infrastructure: Orio
 - <http://tinyurl.com/OrioTool>
 - Extensible design, few prerequisites, portable



```
orcc -spec=foo-bgp.spec -output=foo.bgp.c foo.c
```

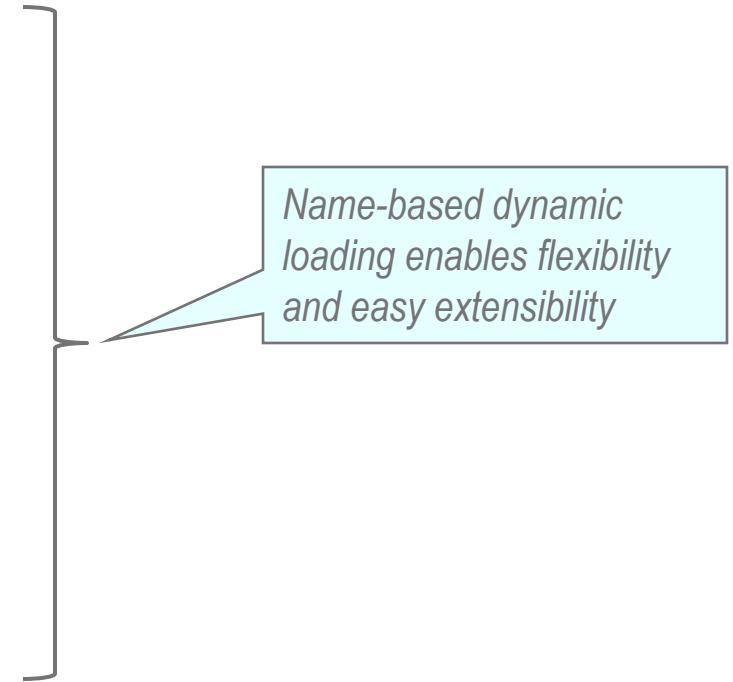
Orio Workflow



Orio Design

- Orio is implemented in Python and consists of

- Optimization modules that provide
 - Annotation parsing
 - Transformation
 - Code generation
 - Modules can contain submodules.
- Empirical tuning modules that provide
 - Tuning spec parsing
 - Search space exploration
 - Test execution (sequential and parallel)



Search Engine

- Inputs:
 - Transformation specification
 - Performance experiment definition
 - Program inputs
 - Performance parameters
 - Constraints
 - Search method
 - Build and execution environment description
- Current search strategies:
 - Exhaustive
 - Random
 - Nelder-Mead Simplex algorithm with a time limit constraint
 - Simulated annealing
 - Derivative-free global optimization (in progress)



Experiments

- **Blue Gene/P** (four 850 MHz IBM PowerPC 450 processors with a dual floating-point unit and 2 GB total memory per node, private L1 (32 KB) and L2 (4 MB) caches, a shared L3 cache (8 MB))
- **Xeon workstation** (dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64))
- **Transformations**
 - Simple loop unrolling
 - Loop unroll/jamming
 - Loop tiling
 - Loop permutation
 - Scalar replacement
 - Register tiling
 - Loop-bound replacement
 - Array copy optimization
 - Multicore parallelization (using OpenMP)
 - SIMD instructions (BG/P)



Streaming Operations

- Stream triad operation:

```
for (i = 0; i < n; i++)  
    z[i] = x[i] + ss * y[i];
```

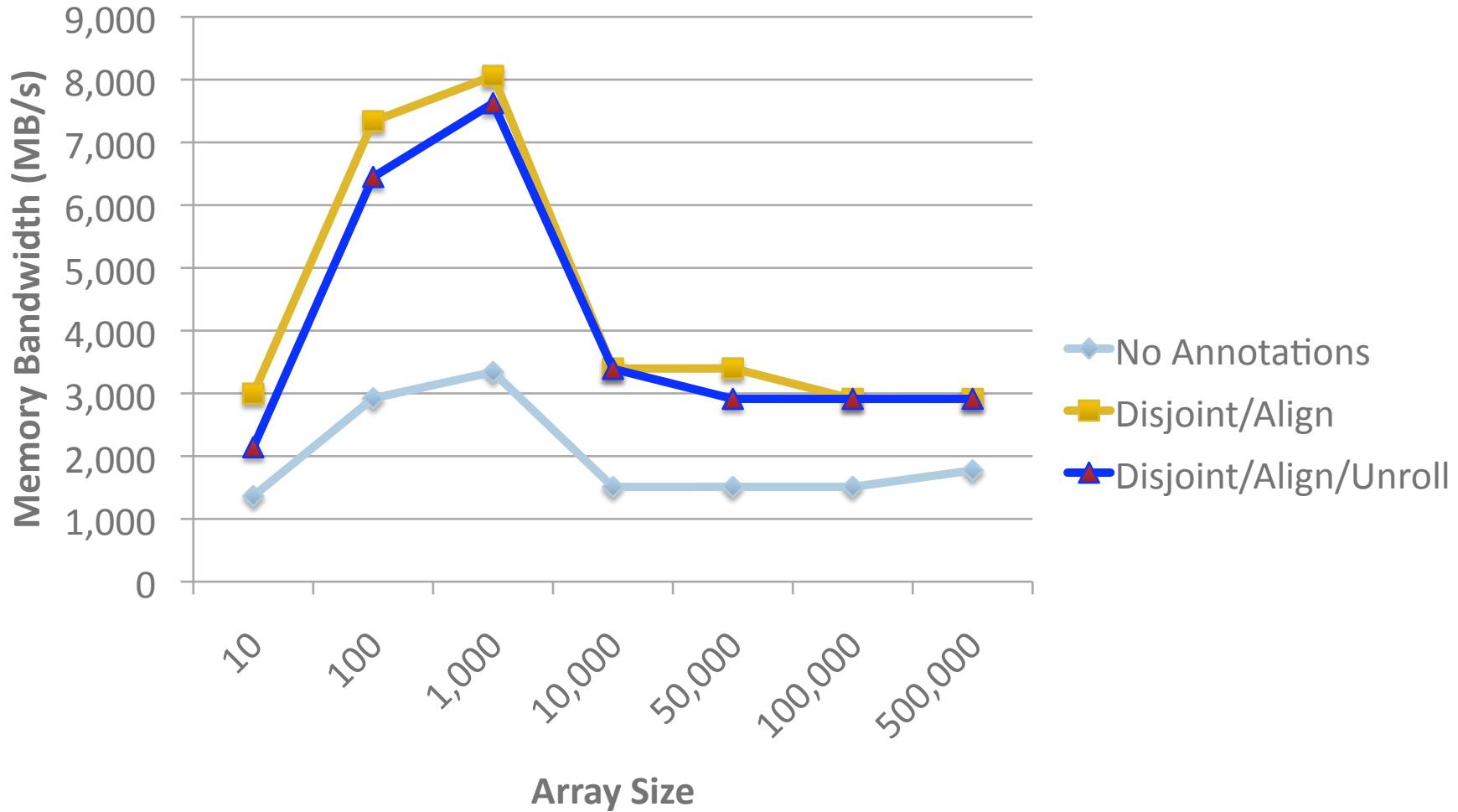
- AXPY-4:

```
for (i=0; i<=N-1; i++)  
    y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
```

*AXPY_N-type operations occur frequently
in derivative codes generated using
automatic differentiation tools.*



Stream Triad Results: BG/P

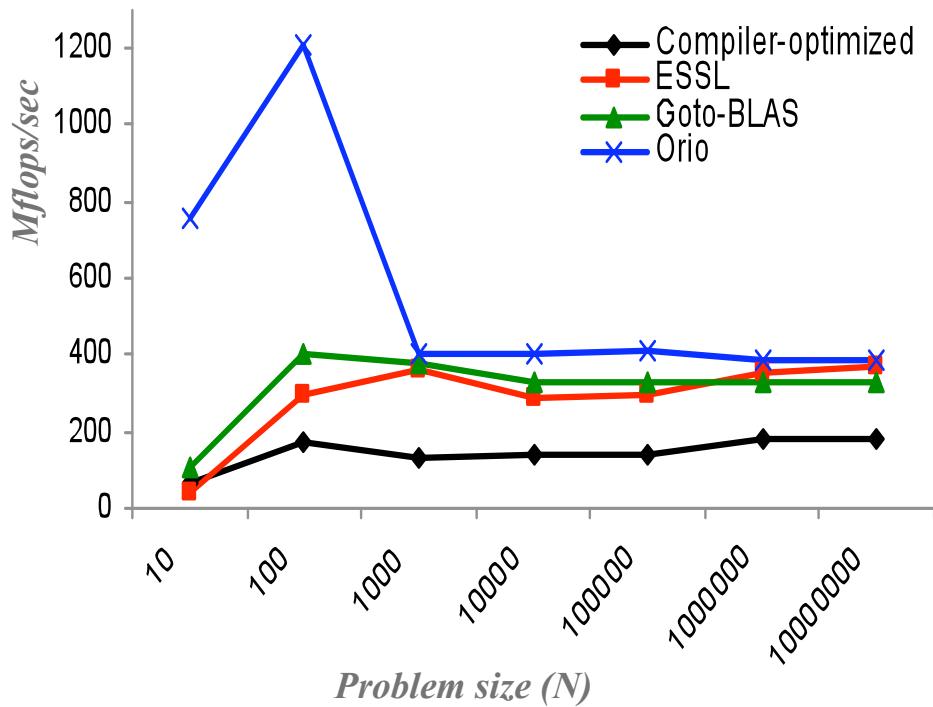


Blue Gene/P

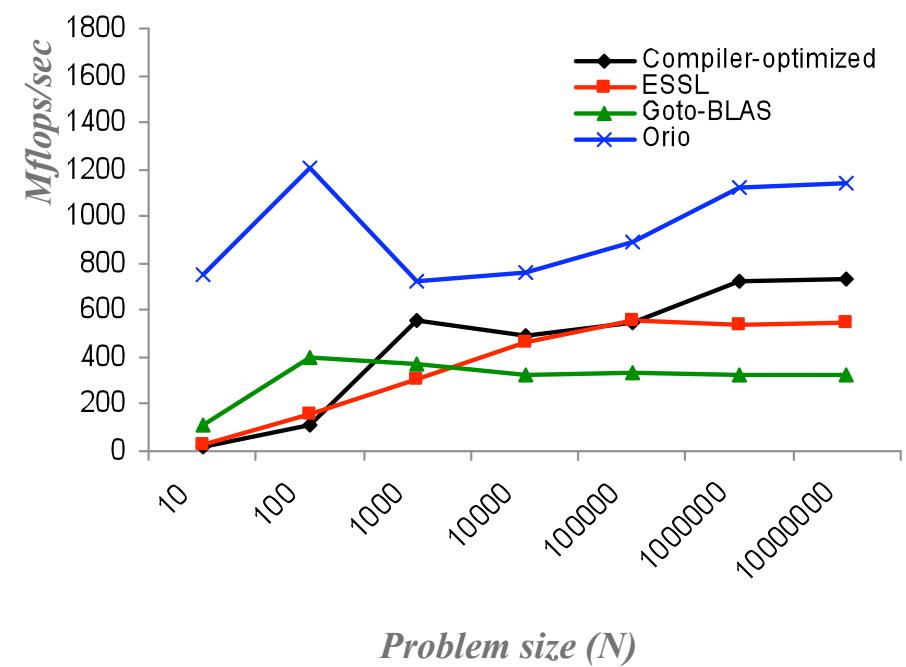


Streaming Operation: AXPY4

```
for (i=0; i<=N-1; i++)  
    y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
```



Sequential



Parallel (four cores)



Blue Gene/P

Stencil Computations and Dense Linear Algebra

- Finite-Difference Time-Domain (FDTD) methods
- Composed dense linear algebra operations

- ATAX

$$y \leftarrow A^T (A x)$$

- GEMVER

$$A \leftarrow A + u_1 v_1 + u_2 v_2$$

$$x \leftarrow \beta A^T y + z$$

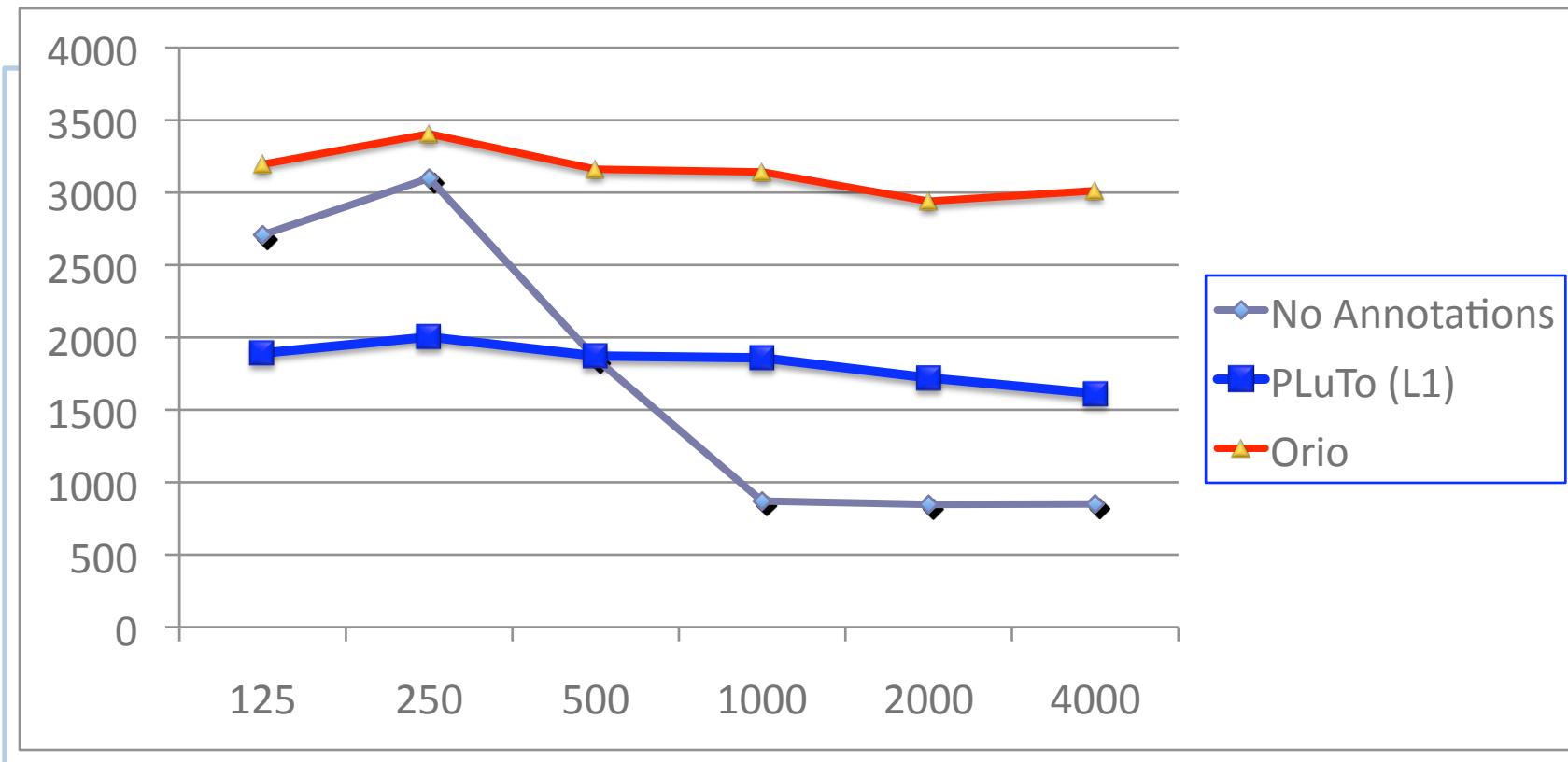
$$w \leftarrow \alpha A x$$



More on
that later



2D FDTD: MFLOPs/sec. vs. Array Size (1 Core, tmax=500)

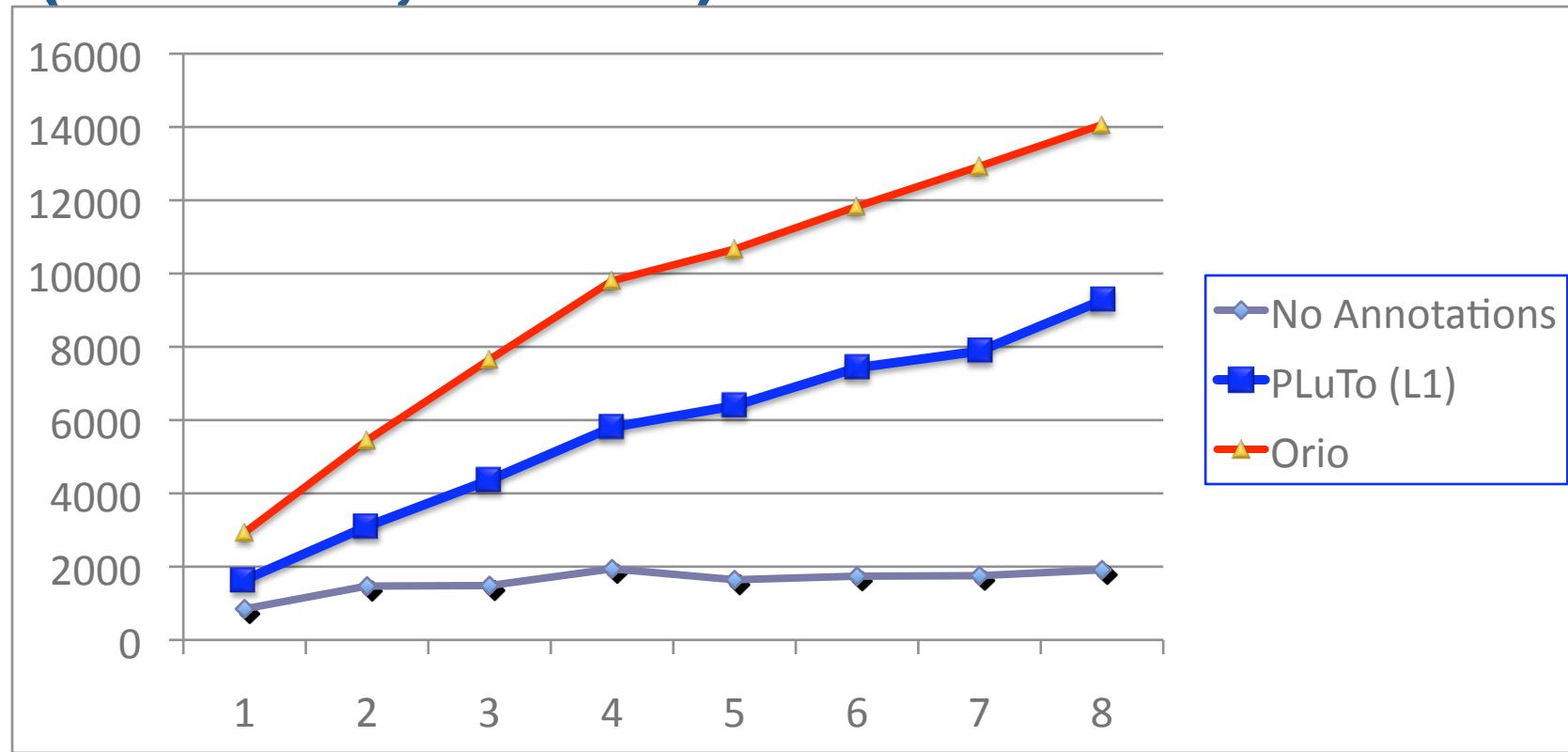


}

⌚ Xeon workstation (dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64))



2D FDTD: MFLOPs/sec. vs. Number of Cores (tmax=500, N=2000)



⌚ Xeon workstation (dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64))



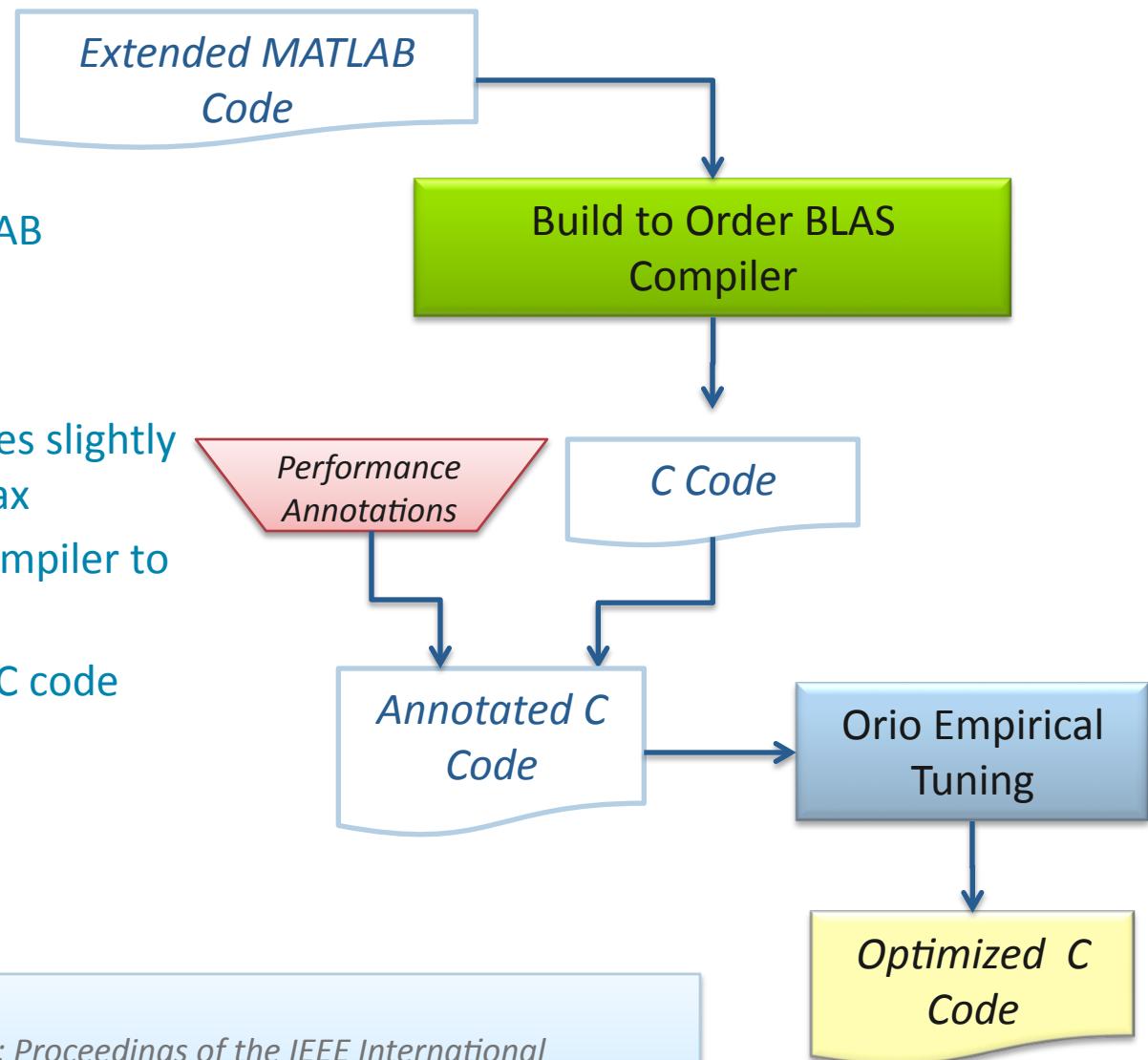
Composed dense linear algebra computations

- Productivity:

- Leverage existing MATLAB compiler technology¹

- Performance:

- Annotation language uses slightly extended MATLAB syntax
 - Use existing MATLAB compiler to generate C code
 - Perform analysis of the C code
 - Tune with Orio



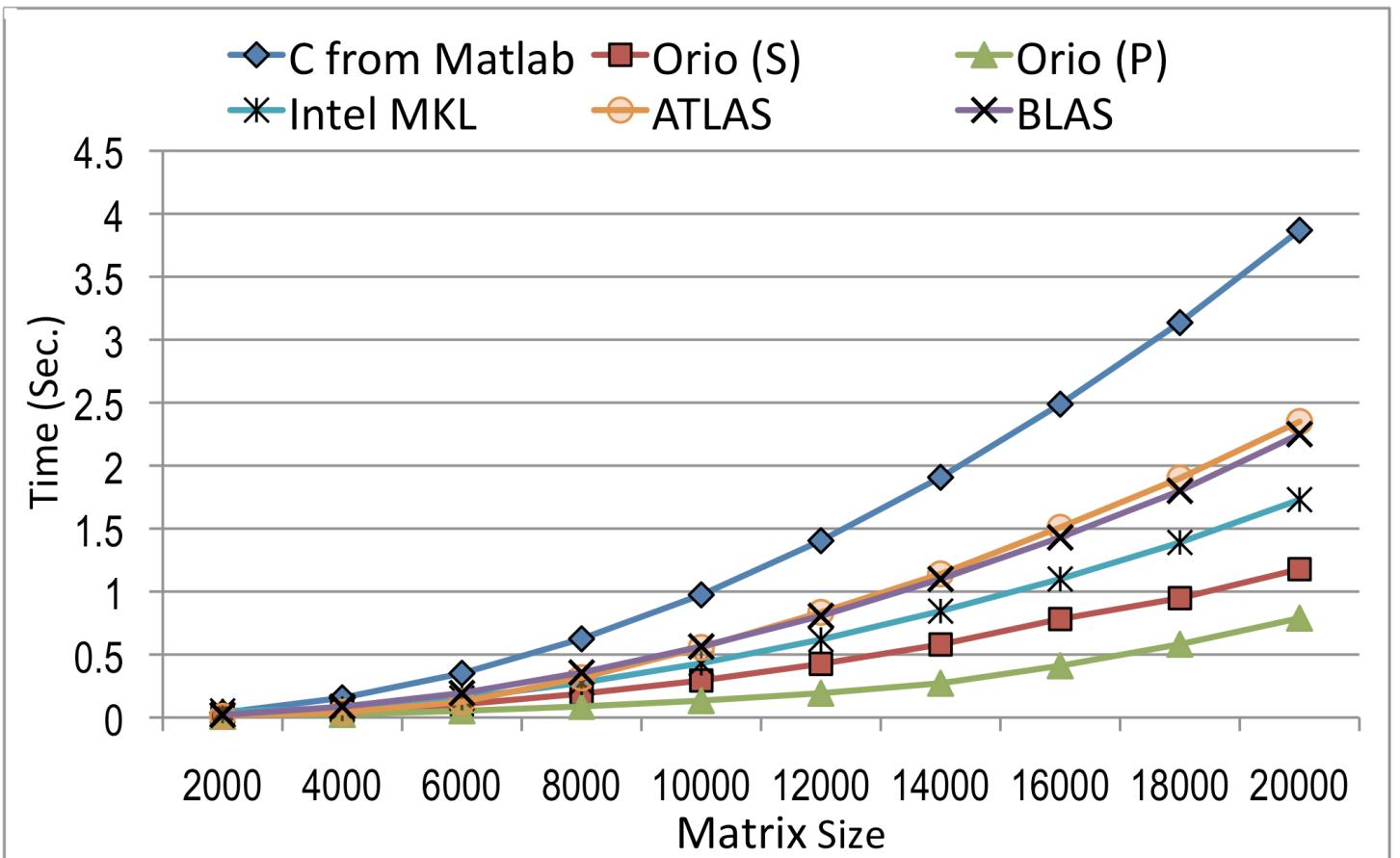
¹Jessup, E., Karlin, I., , Siek, J.:

Build to order linear algebra kernels. In: *Proceedings of the IEEE International Symposium on Parallel and Distributed (IPDPS 2008)*, IEEE (2008) 1–8.



Composed linear algebra: ATAX

```
ATAX
in
A : row matrix,
x : vector
out
y : vector
{
    y = A' * (A * x)
}
```

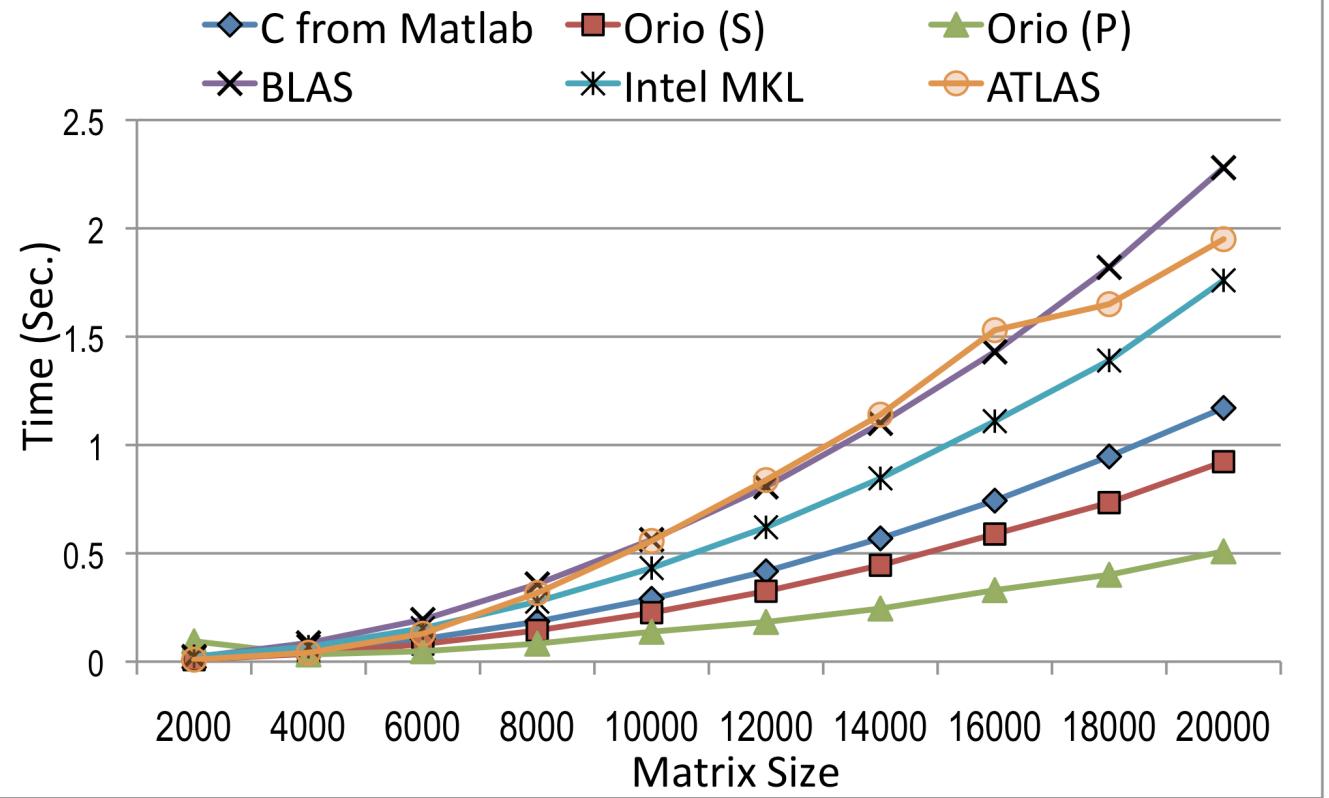


- ⌚ Xeon workstation (dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64))



Composed linear algebra: BiCG Kernel

```
BiCG
in
A : row matrix,
p : vector, r : vector
out
q : vector, s : vector
{
    q = A * p
    s = A' * r
}
```

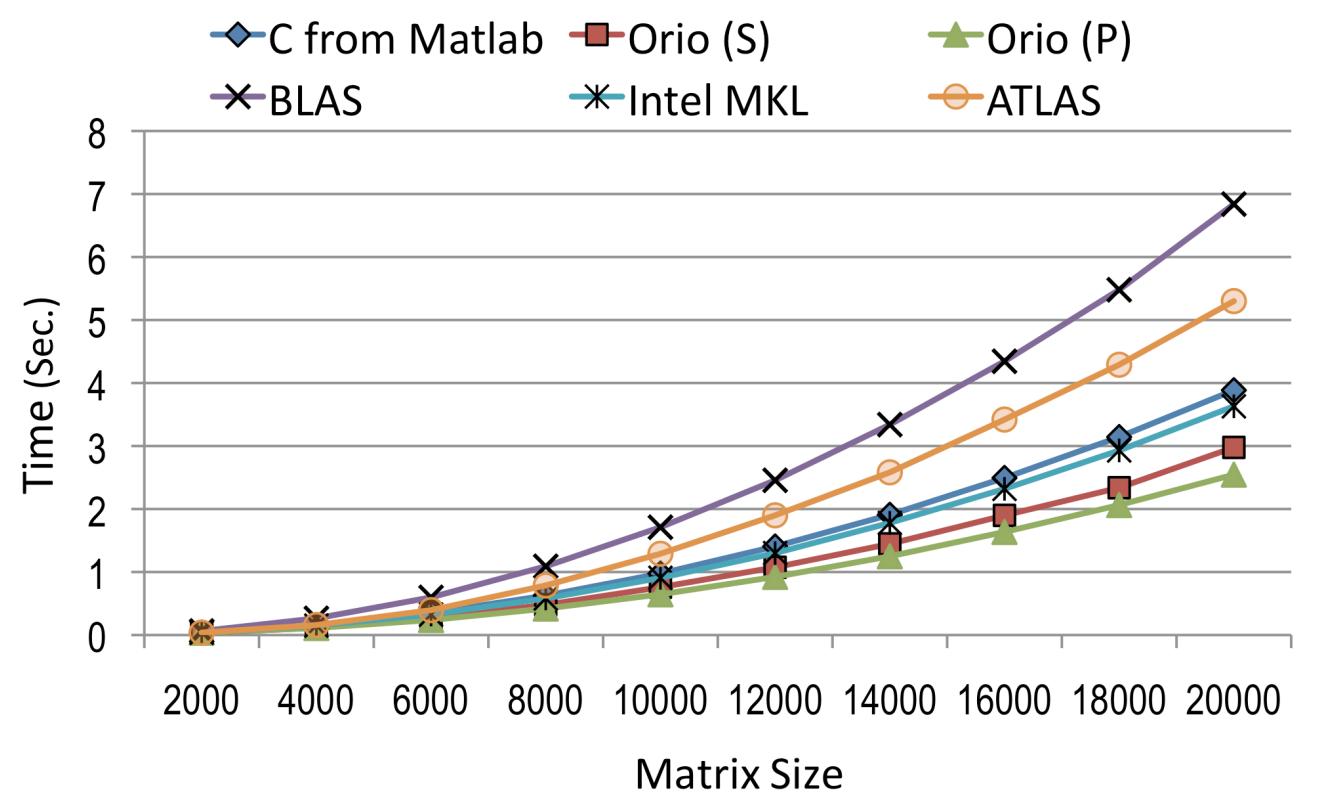


⌚ Xeon workstation (dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64))



Composed Linear Algebra: GEMVER

```
GEMVER
in
A : column matrix,
u1 : vector, u2 : vector,
v1 : vector, v2 : vector,
a : scalar, b : scalar,
y : vector, z : vector
out
B : column matrix,
x : vector, w : vector
{
    B = A + u1 * v1'
        + u2 * v2'
    x = b * (B' * y) + z
    w = a * (B * x)
}
```



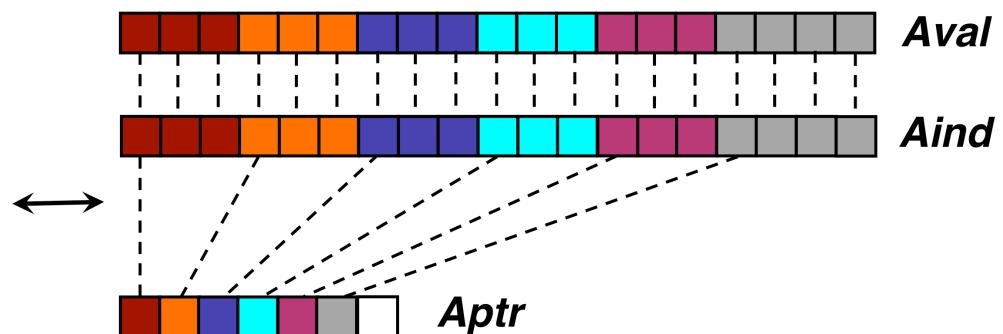
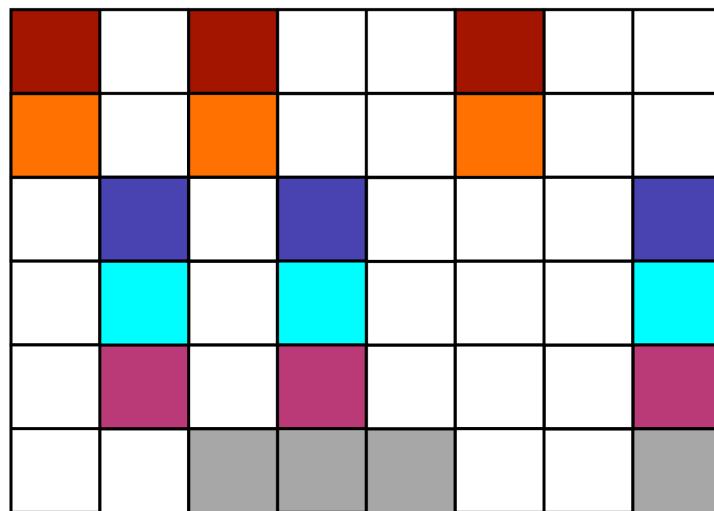
⌚ Xeon workstation (dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64))

Sparse Linear Algebra

- PETSc (<http://www.mcs.anl.gov/petsc>): a toolkit for the parallel numerical solution of partial differential equations
- Sparse matrix-vector multiplication dominates the performance of many applications

$$\forall A_{i,j} \neq 0 : y_i \leftarrow y_i + A_{i,j} \cdot x_j$$

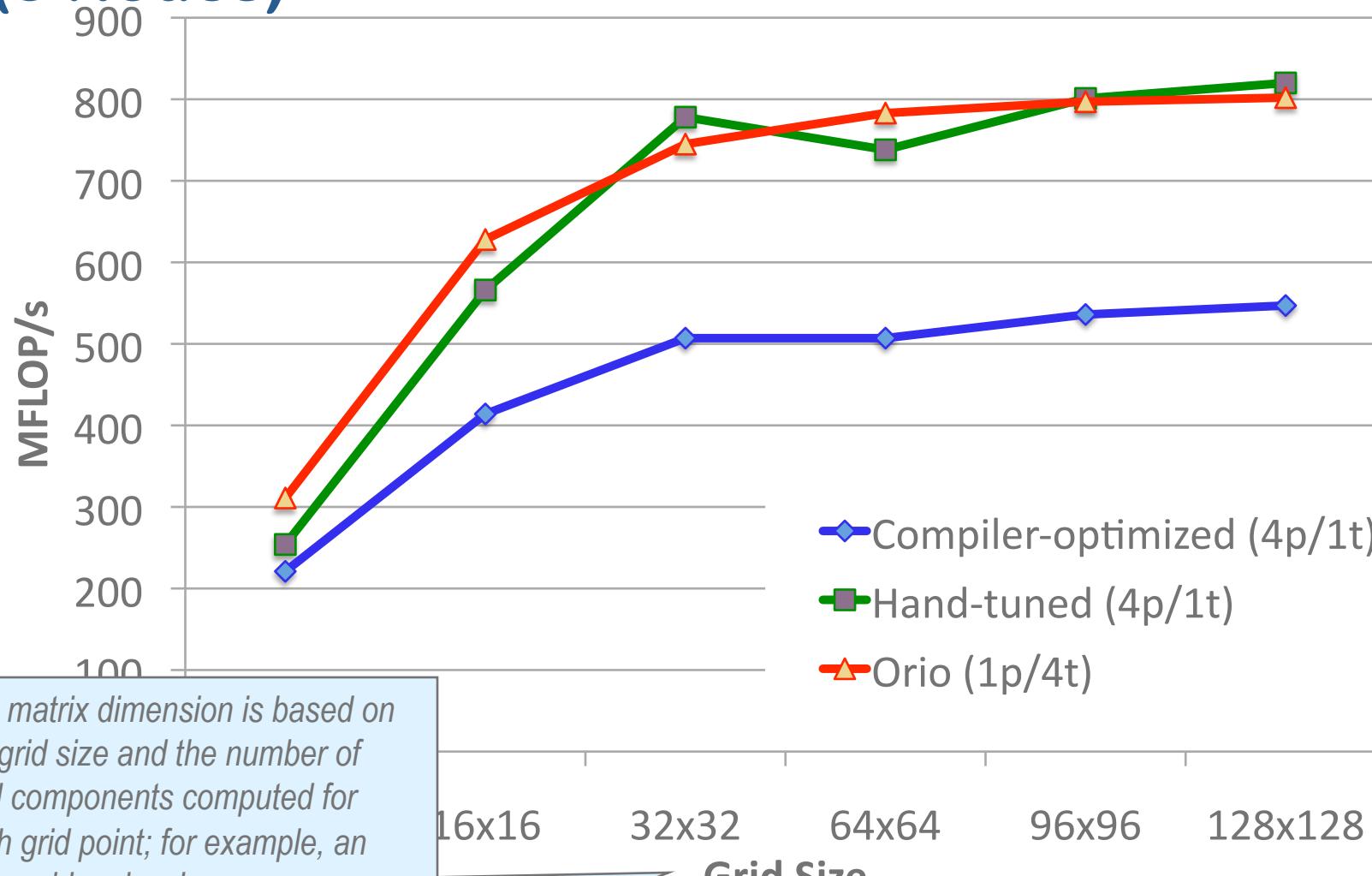
where A is a sparse matrix and x, y are dense vectors.



```
for (i=0; i<num_rows; i++)  
  for (j=Aptra[i]; j<Aptra[i+1]; j++)  
    y[i] += Aval[j]*x[Aind[j]];
```

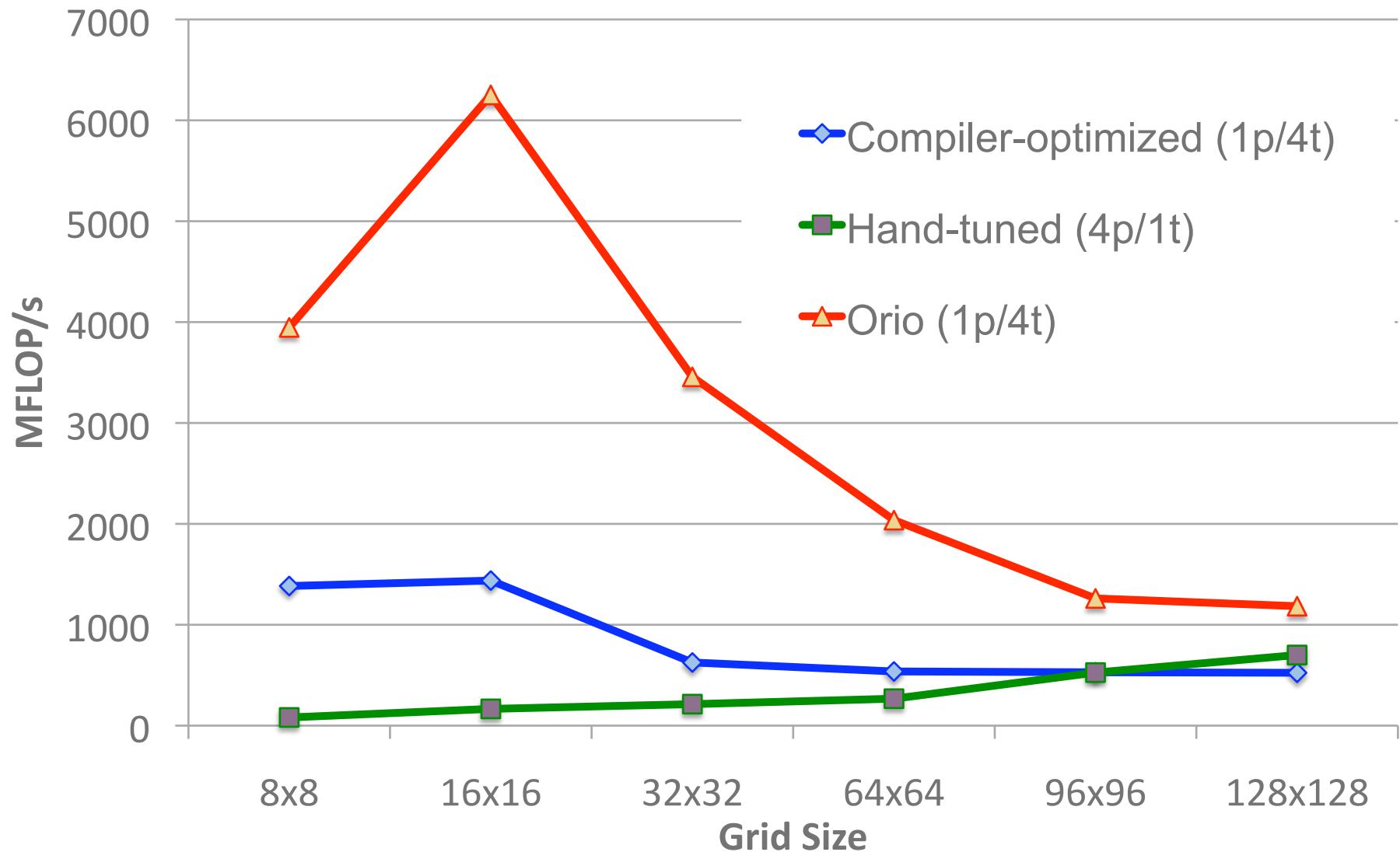


SpMV in 2D Driven Cavity Flow (8 Nodes)



The matrix dimension is based on the grid size and the number of field components computed for each grid point; for example, an **8×8** problem involves sparse matrices of dimension **900** with **17,040** nonzero entries.

SpMV in a 2D driven cavity flow



Summary

- Useful performance information can be generated automatically through source code analysis
 - PBound: <http://tinyurl.com/PBound>
- Both high productivity and high performance can be supported through annotation-based empirical tuning
 - Pbound: <http://tinyurl.com/OrioTool>
- Thank you!

